Back To Basics
API Design

JASON TURNER

Cppcon
The C++ Conference

2022

September 12th-16th

@lefticus

emptycrate.com/idocpp

Jason Turner

- Host of C++ Weekly https://www.youtube.com/c/lefticus1
- Author
  - C++ Best Practices
  - OpCode, Copy and Reference, Object Lifetime Puzzlers
  - https://amzn.to/3xWh8Ox
  - https://leanpub.com/u/jason_turner
- Developer
  - https://cppbestpractices.com
  - https://github.com/cpp-best-practices
- Microsoft MVP for C++ 2015-present

Jason Turner

Independent and available for training and code reviews

- https://articles.emptycrate.com/idocpp

About my Talks

- Move to the front!
- Please interrupt and ask questions
- This is approximately how my training days look

Local Meetups

I got the chance to practice this talk at my meetup.

Local Meetups

I got the chance to practice this talk at my meetup.

And therefor have no idea how long this is going to take.

Local Meetups

I got the chance to practice this talk at my meetup.

And therefor have no idea how long this is going to take.

- Do you live in a city with a local meetup?

Local Meetups

I got the chance to practice this talk at my meetup.

And therefor have no idea how long this is going to take.

- Do you live in a city with a local meetup?
- Do you attend your local meetup?

Local Meetups

I got the chance to practice this talk at my meetup.

And therefor have no idea how long this is going to take.

- Do you live in a city with a local meetup?
- Do you attend your local meetup?
- Do you live or work in the Denver/Boulder/Longmont area?

Local Meetups

I got the chance to practice this talk at my meetup.

And therefor have no idea how long this is going to take.

- Do you live in a city with a local meetup?
- Do you attend your local meetup?
- Do you live or work in the Denver/Boulder/Longmont area?
- Do you attend my meetup?

Back To Basics: API Design

I was asked if I could give a Back To Basics track talk on API design.

Back To Basics: API Design

I was asked if I could give a Back To Basics track talk on API design.

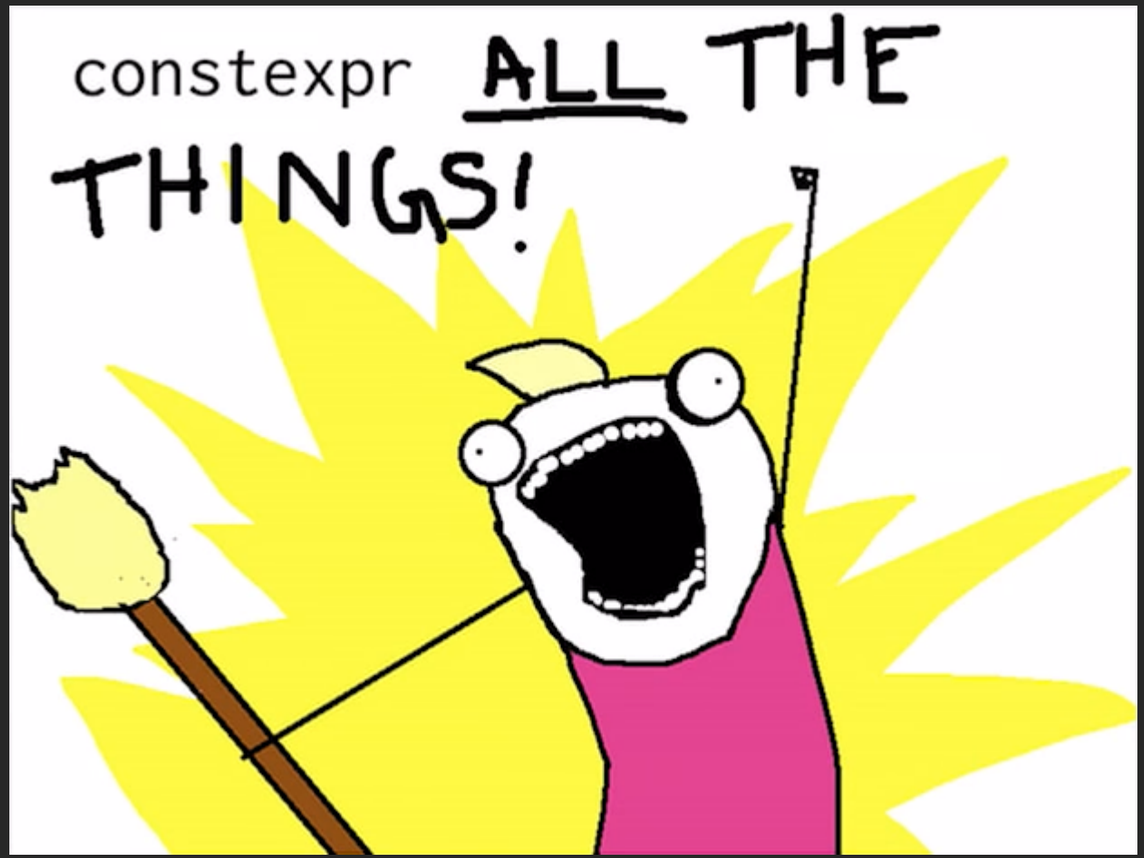
> *Sure, I can do that.*

Back To Basics: API Design

I was asked if I could give a Back To Basics track talk on API design.

> *Sure, I can do that.*

- Me



More Seriously

Back To Basics: API Design

Back To Basics: API Design

- Outside of the kinds of things I tend to talk about for conferences

Back To Basics: API Design

- Outside of the kinds of things I tend to talk about for conferences
- Largely taken from material I normally leave for classes

Back To Basics: API Design

- Outside of the kinds of things I tend to talk about for conferences
- Largely taken from material I normally leave for classes
- You should hire me to come do training at your company!

C++ Best Practices #32: “Make Your API Hard To Use Wrong”

Make Your API Hard To Use Wrong

- This is what we will be focusing on in this session
- I show you code, you tell me if the API is easy or hard to use wrong (or if there is a gray area).

And Maybe Something About `constexpr`

The Common `std::vector` (C++98 version)

```
1  template<typename T>
2  class vector {
3  public:
4    bool empty() const;
5  };
```

https://godbolt.org/z/hsYvEW9sd

Easy or Hard to use wrong? Why?

The Common `std::vector` (C++98 version)

```cpp
template<typename T>
class vector {
public:
  bool empty() const;
};
```

https://godbolt.org/z/hsYvEW9sd

Easy or Hard to use wrong? Why?

- What does `empty()` do?

The Common `std::vector` (C++98 version)

```
1  template<typename T>
2  class vector {
3  public:
4    bool empty() const;
5  };
```

https://godbolt.org/z/hsYvEW9sd

Easy or Hard to use wrong? Why?

- What does `empty()` do?
- What happens if we drop the return value?

The Common `std::vector` (C++98 version)

```
1  template<typename T>
2  class vector {
3  public:
4    bool empty() const;
5  };
```

https://godbolt.org/z/hsYvEW9sd

Easy or Hard to use wrong? Why?

- What does `empty()` do?
- What happens if we drop the return value?
- What kind of error handling does it have?

The Common `std::vector`

```
1  template<typename T>
2  class vector {
3  public:
4    bool empty() const;
5  };
```

https://godbolt.org/z/hsYvEW9sd

How would you rewrite this?

The Common `std::vector`

```cpp
template<typename T>
class vector {
public:
  bool empty() const;
};
```

https://godbolt.org/z/hsYvEW9sd

How would you rewrite this?

```cpp
[[nodiscard]] bool is_empty() const;
```

Easy or Hard to use wrong?

The Common `std::vector`

```
1  template<typename T>
2  class vector {
3  public:
4    bool empty() const;
5  };
```

https://godbolt.org/z/hsYvEW9sd

How would you rewrite this?

```
1  [[nodiscard]] bool is_empty() const;
```

Easy or Hard to use wrong?

What kind of error handling does this have? Are there any reasonable errors for it?

The Common `std::vector`

```
1  template<typename T>
2  class vector {
3  public:
4    [[nodiscard]] bool is_empty() const noexcept;
5  };
```

https://godbolt.org/z/3Ef9oGj9r

Easy or Hard to use wrong?

Use Better Naming

Use Better Naming

Naming is hard.

As we know, the two hardest problems in computer science are:

Use Better Naming

Naming is hard.

As we know, the two hardest problems in computer science are:

1. Cache Invalidation

Use Better Naming

Naming is hard.

As we know, the two hardest problems in computer science are:

1. Cache Invalidation
2. Naming

Use Better Naming

Naming is hard.

As we know, the two hardest problems in computer science are:

1. Cache Invalidation
2. Naming
3. Off-by-one Errors

Use Better Naming

Naming is hard.

As we know, the two hardest problems in computer science are:

1. Cache Invalidation
2. Naming
3. Off-by-one Errors
4. Scope creep

Use Better Naming

Naming is hard.

As we know, the two hardest problems in computer science are:

1. Cache Invalidation
2. Naming
3. Off-by-one Errors
4. Scope creep
5. Bounds checking

Use Better Naming

Naming is hard.

As we know, the two hardest problems in computer science are:

1. Cache Invalidation
2. Naming
3. Off-by-one Errors
4. Scope creep
5. Bounds checking

Phil Karlton, Unknown, Dave Stagner

[[nodiscard]]

`[[nodiscard]]` and Functions

Instructs the compiler to generate a warning if a return value is dropped. Can be applied to types or function declarations.

```
1  [[nodiscard]] int get_value();
2
3  int main() {
4    get_value(); // warning issued from any reasonable compiler
5  }
```

https://godbolt.org/z/xvrjhjGdK

`[[nodiscard]]` and Lambdas

C++23 fixes a minor loophole in the standard and now allows `[[nodiscard]]` with lambdas.

```
1  int main() {
2    auto l = [] [[nodiscard]] () -> int { return 42; };
3
4    l(); // warning here
5  }
```

https://godbolt.org/z/ad444nqqW

[[nodiscard]] and Types

```
1  struct [[nodiscard]] ErrorType{};
2  ErrorType get_value();
3
4  int main() {
5    get_value(); // warning issued from any reasonable compiler
6  }
```

https://godbolt.org/z/Gdv8YsMcG

`[[nodiscard]]` and Constructors

```
1  struct FDHolder {
2    [[nodiscard]] FDHolder(int FD);
3    FDHolder();
4  };
5
6  int main() {
7    FDHolder{42};   // warning
8    FDHolder h{42}; // constructed object not discarded, no warning
9    FDHolder{};     // default constructed, no warning
10 }
```

https://godbolt.org/z/x5e87r7Ka

`[[nodiscard]]` Summary

`[[nodiscard]]` Summary

- Used to indicate when it is an error to ignore a return value from a function

`[[nodiscard]]` Summary

- Used to indicate when it is an error to ignore a return value from a function
- Can be applied to constructors as of C++20

`[[nodiscard]]` Summary

- Used to indicate when it is an error to ignore a return value from a function
- Can be applied to constructors as of C++20
- Can have a message to explain the error `[[nodiscard("Lock objects should never be discarded")]]`

`[[nodiscard]]` Summary

- Used to indicate when it is an error to ignore a return value from a function
- Can be applied to constructors as of C++20
- Can have a message to explain the error `[[nodiscard("Lock objects should never be discarded")]]`
- Should be used extensively. Any non-mutating (getter/accessor/`const`) function should be `[[nodiscard]]`

`[[nodiscard]]` Summary

- Used to indicate when it is an error to ignore a return value from a function
- Can be applied to constructors as of C++20
- Can have a message to explain the error `[[nodiscard("Lock objects should never be discarded")]]`
- Should be used extensively. Any non-mutating (getter/accessor/`const`) function should be `[[nodiscard]]`
  - `cos()`, `[[nodiscard]]`?

`[[nodiscard]]` Summary

- Used to indicate when it is an error to ignore a return value from a function
- Can be applied to constructors as of C++20
- Can have a message to explain the error `[[nodiscard("Lock objects should never be discarded")]]`
- Should be used extensively. Any non-mutating (getter/accessor/`const`) function should be `[[nodiscard]]`
  - `cos()`, `[[nodiscard]]`?
  - `vector::insert()`, `[[nodiscard]]`?

`[[nodiscard]]` Summary

- Used to indicate when it is an error to ignore a return value from a function
- Can be applied to constructors as of C++20
- Can have a message to explain the error `[[nodiscard("Lock objects should never be discarded")]]`
- Should be used extensively. Any non-mutating (getter/accessor/`const`) function should be `[[nodiscard]]`
  - `cos()`, `[[nodiscard]]`?
  - `vector::insert()`, `[[nodiscard]]`?
- Can be checked / enforced with static analysis

`noexcept`

noexcept

`noexcept` notifies the user (and compiler) that a function may not throw an exception. If an exception is thrown from that function, `terminate` MUST be called.

```
1  void myfunc() noexcept {
2    // required to terminate the program
3    throw 42;
4  }
5
6  int main() {
7    try {
8      myfunc();
9    } catch(...) {
10     // catch is irrelevant, `terminate` is called
11   }
12 }
```

https://godbolt.org/z/P1EjKbMsc

Summary So Far

Summary So Far

Summary So Far

- *Use better naming*

Summary So Far

- *Use better naming*
- *Use* `[[nodiscard]]` *(with reasons) liberally*

Summary So Far

- *Use better naming*
- *Use* `[[nodiscard]]` *(with reasons) liberally*
- *Use* `noexcept` *to help indicate what kind of error handling is being used*

Increasing The Stakes: A Factory Function

A Factory Function

```
1 | Widget *make_widget(int widget_type);
```

Easy or Hard to use wrong?

A Factory Function

```
1 | Widget *make_widget(int widget_type);
```

Easy or Hard to use wrong?

- What happens if we ignore the return value?

A Factory Function

```
1 | Widget *make_widget(int widget_type);
```

Easy or Hard to use wrong?

- What happens if we ignore the return value?
- What is the possible range of input values?

A Factory Function

```
1 | Widget *make_widget(int widget_type);
```

How would you rewrite this?

A Factory Function

```
1 | Widget *make_widget(int widget_type);
```

How would you rewrite this?

```
1 | [[nodiscard]] std::unique_ptr<Widget> make_widget(int widget_type);
```

Easy or Hard to use wrong?

A Factory Function

```
1  enum class WidgetType {
2    Slider = 0,
3    Button = 1
4  };
```

```
1  [[nodiscard]] std::unique_ptr<Widget> make_widget(WidgetType type);
```

Easy or Hard to use wrong?

A Factory Function

```cpp
enum class WidgetType {
  Slider = 0,
  Button = 1
};
```

```cpp
[[nodiscard]] std::unique_ptr<Widget> make_widget(WidgetType type);
```

Easy or Hard to use wrong?

```cpp
auto widget = make_widget(static_cast<WidgetType>(-42));
```

A Factory Function

```
1 | enum class WidgetType {
2 |   Slider = 0,
3 |   Button = 1
4 | };
```

```
1 | [[nodiscard]] std::unique_ptr<Widget> make_widget(WidgetType type);
```

Easy or Hard to use wrong?

```
1 | auto widget = make_widget(static_cast<WidgetType>(-42));
```

- What about error handling?

A Factory Function

```
1 enum class WidgetType {
2   Slider = 0,
3   Button = 1
4 };
```

```
1 [[nodiscard]] std::unique_ptr<Widget> make_widget(WidgetType type);
```

Easy or Hard to use wrong?

```
1 auto widget = make_widget(static_cast<WidgetType>(-42));
```

- What about error handling?
- Is it possible to fail to create a Widget?

A Factory Function

```
1 | enum class WidgetType {
2 |   Slider = 0,
3 |   Button = 1
4 | };
```

```
1 | [[nodiscard]] std::unique_ptr<Widget> make_widget(WidgetType type);
```

Easy or Hard to use wrong?

```
1 | auto widget = make_widget(static_cast<WidgetType>(-42));
```

- What about error handling?
- Is it possible to fail to create a Widget?
- Should it throw an exception?

A Factory Function

```
1 | enum class WidgetType {
2 |   Slider = 0,
3 |   Button = 1
4 | };
```

```
1 | [[nodiscard]] std::unique_ptr<Widget> make_widget(WidgetType type);
```

Easy or Hard to use wrong?

```
1 | auto widget = make_widget(static_cast<WidgetType>(-42));
```

- What about error handling?
- Is it possible to fail to create a Widget?
- Should it throw an exception?
- Is a `nullptr` return an error condition?

A Factory Function

```
1 | enum class WidgetType {
2 |   Slider = 0,
3 |   Button = 1
4 | };
```

```
1 | [[nodiscard]] std::unique_ptr<Widget> make_widget(WidgetType type);
```

Easy or Hard to use wrong?

```
1 | auto widget = make_widget(static_cast<WidgetType>(-42));
```

- What about error handling?
- Is it possible to fail to create a Widget?
- Should it throw an exception?
- Is a `nullptr` return an error condition?

We'll come back to this in a minute.

Never Return a Raw Pointer

Never Return a Raw Pointer

Never Return a Raw Pointer

- It simply raises too many questions. Who owns it? Who deletes it? Is it a singleton global?

Never Return a Raw Pointer

- It simply raises too many questions. Who owns it? Who deletes it? Is it a singleton global?
- Consider `owning_ptr`, `non_owning_ptr` or some kind of wrapper to document intent, if you must.

Have a Consistent Error Handling Policy

Consistent Error Handling

Consistent Error Handling

- Use one consistent method of reporting errors in your library

Consistent Error Handling

- Use one consistent method of reporting errors in your library
- Strongly avoid out-of-band error reporting (`get_last_error()` or `errno`)

Consistent Error Handling

- Use one consistent method of reporting errors in your library
- Strongly avoid out-of-band error reporting (`get_last_error()` or `errno`)
- Make errors impossible to ignore (no returning an error code!)

Consistent Error Handling

- Use one consistent method of reporting errors in your library
- Strongly avoid out-of-band error reporting (`get_last_error()` or `errno`)
- Make errors impossible to ignore (no returning an error code!)
- Never use `std::optional<>` to indicate an error condition. (it does not convey a reason, and the reason becomes out of bound).

Consistent Error Handling

- Use one consistent method of reporting errors in your library
- Strongly avoid out-of-band error reporting (`get_last_error()` or `errno`)
- Make errors impossible to ignore (no returning an error code!)
- Never use `std::optional<>` to indicate an error condition. (it does not convey a reason, and the reason becomes out of bound).
- Consider `std::expected<>` (C++23) or similar

Summary So Far

Summary So Far

- Use better naming
- Use `[[nodiscard]]` (with reasons) liberally
- Use `noexcept` to help indicate what kind of error handling is being used

Summary So Far

- Use better naming
- Use `[[nodiscard]]` (with reasons) liberally
- Use `noexcept` to help indicate what kind of error handling is being used
- *Never return a raw pointer*

Summary So Far

- Use better naming
- Use `[[nodiscard]]` (with reasons) liberally
- Use `noexcept` to help indicate what kind of error handling is being used
- *Never return a raw pointer*
- *Provide consistent, impossible to ignore error handling with in-band reporting of what went wrong*

The Humble `fopen` Function

```
1 | FILE *fopen(const char *pathname, const char *mode);
```

Easy or Hard to use wrong? Why?

The Humble `fopen` Function

```
1 | FILE *fopen(const char *pathname, const char *mode);
```

Easy or Hard to use wrong? Why?

- How is error handling done?

The Humble `fopen` Function

```
1 | FILE *fopen(const char *pathname, const char *mode);
```

Easy or Hard to use wrong? Why?

- How is error handling done?
- What happens if we drop the return value?

The Humble `fopen` Function

```
1 | FILE *fopen(const char *pathname, const char *mode);
```

Easy or Hard to use wrong? Why?

- How is error handling done?
- What happens if we drop the return value?
- What is the format for `mode`?

The Humble `fopen` Function

```
1 | FILE *fopen(const char *pathname, const char *mode);
```

Easy or Hard to use wrong? Why?

- How is error handling done?
- What happens if we drop the return value?
- What is the format for `mode`?
- What happens if I call `fopen("w", "/my/file/path")`?

The Humble `fopen` Function

```
1 | FILE *fopen(const char *pathname, const char *mode);
```

Easy or Hard to use wrong? Why?

- How is error handling done?
- What happens if we drop the return value?
- What is the format for `mode`?
- What happens if I call `fopen("w", "/my/file/path")`?
- What happens if I call `fopen("/my/file/path", 0)`?

The Humble `fopen` Function

```
1 | FILE *fopen(const char *pathname, const char *mode);
```

How would you rewrite this?

The Humble `fopen` Function

```
1 | FILE *fopen(const char *pathname, const char *mode);
```

How would you rewrite this?

```
1 | using FilePtr = std::unique_ptr<FILE, decltype([](FILE *f) { fclose(f); })>;
2 |
3 | [[nodiscard]] FilePtr fopen(const char *pathname, const char *mode);
```

The Humble `fopen` Function

```
1  using FilePtr = std::unique_ptr<FILE, decltype([](FILE *f) { fclose(f); })>;
2
3  [[nodiscard]] FilePtr fopen(const char *pathname, const char *mode);
```

Easy or Hard to use wrong?

The Humble `fopen` Function

```
1  using FilePtr = std::unique_ptr<FILE, decltype([](FILE *f) { fclose(f); })>;
2
3  [[nodiscard]] FilePtr fopen(const char *pathname, const char *mode);
```

Easy or Hard to use wrong?

Avoid easily swappable parameters:

```
1  using FilePtr = std::unique_ptr<FILE, decltype([](FILE *f) { fclose(f); })>;
2
3  [[nodiscard]] FilePtr fopen(const std::filesystem::path &path,
4                              std::string_view mode);    https://godbolt.org/z/rb7TvhGc9
```

Avoid Easily Swappable Parameters

Avoid Easily Swappable Parameters

Avoid Easily Swappable Parameters

- Two (or more) parameters beside each other of the same type are easy to swap.

Avoid Easily Swappable Parameters

- Two (or more) parameters beside each other of the same type are easy to swap.
- clang-tidy has [bugprone-easily-swappable-parameters]

The Humble `fopen` Function

```cpp
using FilePtr = std::unique_ptr<FILE, decltype([](FILE *f) { fclose(f); })>;

[[nodiscard]] FilePtr fopen(const std::filesystem::path &path,
                            std::string_view mode);
```

https://godbolt.org/z/rb7TvhGc9

Easy or Hard to use wrong?

The Humble `fopen` Function

```cpp
using FilePtr = std::unique_ptr<FILE, decltype([](FILE *f) { fclose(f); })>;

[[nodiscard]] FilePtr fopen(const std::filesystem::path &path,
                            std::string_view mode);    https://godbolt.org/z/rb7TvhGc9
```

Easy or Hard to use wrong?

```cpp
// Unfortunately this still compiles.
auto file = fopen("rw+", "/my/file");
```

The Humble `fopen` Function

```cpp
using FilePtr = std::unique_ptr<FILE, decltype([](FILE *f) { fclose(f); })>;

[[nodiscard]] FilePtr fopen(const std::filesystem::path &path,
                            std::string_view mode);
```

https://godbolt.org/z/rb7TvhGc9

What is the fundamental problem here?

The Humble `fopen` Function

```cpp
using FilePtr = std::unique_ptr<FILE, decltype([](FILE *f) { fclose(f); })>;

[[nodiscard]] FilePtr fopen(const std::filesystem::path &path,
                            std::string_view mode);
```

https://godbolt.org/z/rb7TvhGc9

What is the fundamental problem here?

```cpp
// simplified
namespace std {
  namespace filesystem {
    path(string_type&& source, format fmt=auto_format);
  }
  struct string_view {
    string_view(const char *);
  };
}
```

https://godbolt.org/z/z6G5z3s4s

The Humble `fopen` Function

```cpp
using FilePtr = std::unique_ptr<FILE, decltype([](FILE *f) { fclose(f); })>;

[[nodiscard]] FilePtr fopen(const std::filesystem::path &path,
                            std::string_view mode);
```

https://godbolt.org/z/rb7TvhGc9

What is the fundamental problem here?

```cpp
// simplified
namespace std {
  namespace filesystem {
    path(string_type&& source, format fmt=auto_format);
  }
  struct string_view {
    string_view(const char *);
  };
}
```

https://godbolt.org/z/z6G5z3s4s

Implicit conversions.

Avoid Implicit Conversions / Use Strong Types

Avoid Implicit Conversions / Use Strong Types

Avoid Implicit Conversions / Use Strong Types

- `std::filesystem::path` and `std::string_view` appear to be strongly typed but are not

Avoid Implicit Conversions / Use Strong Types

- `std::filesystem::path` and `std::string_view` appear to be strongly typed but are not
- Implicit conversions between `const char *`, `string`, `string_view`, and `path` break type safety

Avoid Implicit Conversions / Use Strong Types

- `std::filesystem::path` and `std::string_view` appear to be strongly typed but are not
- Implicit conversions between `const char *`, `string`, `string_view`, and `path` break type safety
- Conversion operators and single parameter constructors (including variadic and ones with default parameters) should be `explicit`

The Humble `fopen` Function

```
1  using FilePtr = std::unique_ptr<FILE, decltype([](FILE *f) { fclose(f); })>;
2
3  [[nodiscard]] FilePtr fopen(const std::filesystem::path &path,
4                              std::string_view mode);    https://godbolt.org/z/rb7TvhGc9
```

Assuming `std::filesystem::path` and `std::string_view` are the most correct types for this use case, can we make this better?

The Humble `fopen` Function

```cpp
using FilePtr = std::unique_ptr<FILE, decltype([](FILE *f) { fclose(f); })>;

[[nodiscard]] FilePtr fopen(const std::filesystem::path &path,
                            std::string_view mode);    https://godbolt.org/z/rb7TvhGc9
```

Assuming `std::filesystem::path` and `std::string_view` are the most correct types for this use case, can we make this better?

```cpp
void fopen(const auto &, const auto &) = delete;
```

`=delete` Problematic Overloads

`=delete` Problematic Overloads

`=delete` Problematic Overloads

- Any function can be `=delete`d.

`=delete` Problematic Overloads

- Any function can be `=delete`d.
- If you `=delete` a template, it will become the match for any non-exact parameters, and prevent implicit conversions

Summary So Far

Summary So Far

- Use better naming
- Use `[[nodiscard]]` (with reasons) liberally
- Never return a raw pointer
- Use `noexcept` to help indicate what kind of error handling is being used
- Provide consistent, impossible to ignore error handling with in-band reporting of what went wrong

Summary So Far

- Use better naming
- Use `[[nodiscard]]` (with reasons) liberally
- Never return a raw pointer
- Use `noexcept` to help indicate what kind of error handling is being used
- Provide consistent, impossible to ignore error handling with in-band reporting of what went wrong
- *Use stronger types*

Summary So Far

- Use better naming
- Use `[[nodiscard]]` (with reasons) liberally
- Never return a raw pointer
- Use `noexcept` to help indicate what kind of error handling is being used
- Provide consistent, impossible to ignore error handling with in-band reporting of what went wrong
- *Use stronger types*
- *Avoid default conversions*

Summary So Far

- Use better naming
- Use `[[nodiscard]]` (with reasons) liberally
- Never return a raw pointer
- Use `noexcept` to help indicate what kind of error handling is being used
- Provide consistent, impossible to ignore error handling with in-band reporting of what went wrong
- *Use stronger types*
- *Avoid default conversions*
- *(Sparingly) delete problematic overloads / prevent conversions*

Backing Up a Bit

Making The Factory a Bit Better

With what we now have, can we make this better / harder to use wrong?

```
1 | [[nodiscard]] std::unique_ptr<Widget> make_widget(int widget_type);
```

Making The Factory a Bit Better

With what we now have, can we make this better / harder to use wrong?

```
1 | [[nodiscard]] std::unique_ptr<Widget> make_widget(int widget_type);
```

Depending on context, we might be able to use stronger typing to make our factory better:

```
1 | template<typename WidgetType>
2 | [[nodiscard]] WidgetType make_widget()
3 |   requires (std::is_base_of_v<Widget, WidgetType>);
```

The Humble `fopen` Function

```
1  using FilePtr = std::unique_ptr<FILE, decltype([](FILE *f) { fclose(f); })>;
2
3  [[nodiscard]] FilePtr fopen(const char *pathname, const char *mode);
```

The Humble `fopen` Function

```
1  using FilePtr = std::unique_ptr<FILE, decltype([](FILE *f) { fclose(f); })>;
2
3  [[nodiscard]] FilePtr fopen(const char *pathname, const char *mode);
```

- Are `pathname` and `mode` optional?

The Humble `fopen` Function

```cpp
using FilePtr = std::unique_ptr<FILE, decltype([](FILE *f) { fclose(f); })>;

[[nodiscard]] FilePtr fopen(const char *pathname, const char *mode);
```

- Are `pathname` and `mode` optional?
- What happens if `nullptr` is passed to either of them?

The Humble `fopen` Function

```
1  using FilePtr = std::unique_ptr<FILE, decltype([](FILE *f) { fclose(f); })>;
2
3  [[nodiscard]] FilePtr fopen(const char *pathname, const char *mode);
```

- Are `pathname` and `mode` optional?
- What happens if `nullptr` is passed to either of them?
- UB is invoked

The Humble `fopen` Function

```
1  using FilePtr = std::unique_ptr<FILE, decltype([](FILE *f) { fclose(f); })>;
2
3  [[nodiscard]] FilePtr fopen(const char *pathname, const char *mode);
```

- Are `pathname` and `mode` optional?
- What happens if `nullptr` is passed to either of them?
- UB is invoked

Easy or Hard to use wrong (in this regard?)

Only Pass Raw Pointers for Single Optional Objects

Only Pass Raw Pointers for Single Optional Objects

```cpp
#include <cassert>
#include <string>

void use_string(std::string * const * str) {
  assert(str != nullptr); // is str optional?
  // do things
}
```

https://godbolt.org/z/8ed6b4rTj

Only Pass Raw Pointers for Single Optional Objects

```cpp
#include <string>

void use_string(std::string * const * str) {
  if (str) { // is str optional?
    // do things
  } else {
    // do other things
  }
}
```

https://godbolt.org/z/xMEdbvE59

Only Pass Raw Pointers for Single Optional Objects

If you pass a pointer, you must check it for `nullptr`.

```
1  #include <string>
2
3  void use_string(std::string const * const str) {
4    puts(str->c_str()); // do not do, unsafe
5  }
```

https://godbolt.org/z/4dx5oz1bz

Prefer & Parameters For Non-Small, Non-Trivial Objects

```cpp
#include <string>

// non-trivial, pass by (const) reference
void use_string(const std::string &str) {
  puts(str.c_str());
}

void use_int(const int i) { // trivial and small, copy it
  // use i.
}
```

https://godbolt.org/z/x36G1ssoc

Don’t Pass Smart Pointers Unless You Need to Participate In The Lifetime

Avoiding Passing Smart Pointers

(This is a much bigger discussion)

```
1  #include <string>
2  #include <memory>
3
4  // API usage is artificially limited
5  void use_string(const std::shared_ptr<const std::string> &str) {
6    if (str) { // we don't care it's a
7               // shared pointer, we're treating as a raw *!
8      // use string
9    }
10 }
```

https://godbolt.org/z/W9E13eGxc

Summary So Far

Summary So Far

- Use better naming
- Use `[[nodiscard]]` (with reasons) liberally
- Never return a raw pointer
- Use `noexcept` to help indicate what kind of error handling is being used
- Provide consistent, impossible to ignore error handling with in-band reporting of what went wrong
- Use stronger types
- Avoid default conversions
- (Sparingly) delete problematic overloads / prevent conversions

Summary So Far

- Use better naming
- Use `[[nodiscard]]` (with reasons) liberally
- Never return a raw pointer
- Use `noexcept` to help indicate what kind of error handling is being used
- Provide consistent, impossible to ignore error handling with in-band reporting of what went wrong
- Use stronger types
- Avoid default conversions
- (Sparingly) delete problematic overloads / prevent conversions
- *Avoid passing pointers (only to be used for single/optional objects)*

Summary So Far

- Use better naming
- Use `[[nodiscard]]` (with reasons) liberally
- Never return a raw pointer
- Use `noexcept` to help indicate what kind of error handling is being used
- Provide consistent, impossible to ignore error handling with in-band reporting of what went wrong
- Use stronger types
- Avoid default conversions
- (Sparingly) delete problematic overloads / prevent conversions
- *Avoid passing pointers (only to be used for single/optional objects)*
- *Avoid passing smart pointers*

One Elephant Left In This Example

The Humble `fopen` Function

```cpp
using FilePtr = std::unique_ptr<FILE, decltype([](FILE *f) { fclose(f); })>;

[[nodiscard]] FilePtr fopen(const std::filesystem::path &path,
                            std::string_view mode);
```

https://godbolt.org/z/rb7TvhGc9

What is the possible set of inputs to `mode`?

The Humble `fopen` Function

```cpp
using FilePtr = std::unique_ptr<FILE, decltype([](FILE *f) { fclose(f); })>;

[[nodiscard]] FilePtr fopen(const std::filesystem::path &path,
                            std::string_view mode);
```

https://godbolt.org/z/rb7TvhGc9

What is the possible set of inputs to `mode`?

> *When parsing for individual flag characters in mode (i.e., the characters preceding the "ccs" specification), the glibc implementation of `fopen()` and `freopen()` limits the number of characters examined in mode to 7 (or, in glibc versions before 2.14, to 6, which was not enough to include possible specifications such as `"rb+cmxe"`). The current implementation of `fdopen()` parses at most 5 characters in mode.*

We Can Always Count On POSIX APIs For Interesting Discussion

The Humble `fopen` Function

```cpp
using FilePtr = std::unique_ptr<FILE, decltype([](FILE *f) { fclose(f); })>;

[[nodiscard]] FilePtr fopen(const std::filesystem::path &path,
                            std::string_view mode);
```

https://godbolt.org/z/rb7TvhGc9

Easy or Hard to use wrong?

The Humble `fopen` Function

```cpp
using FilePtr = std::unique_ptr<FILE, decltype([](FILE *f) { fclose(f); })>;

[[nodiscard]] FilePtr fopen(const std::filesystem::path &path,
                            std::string_view mode);
```

https://godbolt.org/z/rb7TvhGc9

Easy or Hard to use wrong?

How do we fix this?

The Humble `fopen` Function

```cpp
using FilePtr = std::unique_ptr<FILE, decltype([](FILE *f) { fclose(f); })>;

[[nodiscard]] FilePtr fopen(const std::filesystem::path &path,
                            std::string_view mode);
```

https://godbolt.org/z/rb7TvhGc9

Easy or Hard to use wrong?

How do we fix this?

- Maybe it's possible to make this some sort of compile-time type checked set?

The Humble `fopen` Function

```cpp
using FilePtr = std::unique_ptr<FILE, decltype([](FILE *f) { fclose(f); })>;

[[nodiscard]] FilePtr fopen(const std::filesystem::path &path,
                            std::string_view mode);
```

https://godbolt.org/z/rb7TvhGc9

Easy or Hard to use wrong?

How do we fix this?

- Maybe it's possible to make this some sort of compile-time type checked set?
- Maybe some bit flags would work?

The Humble `fopen` Function

```cpp
using FilePtr = std::unique_ptr<FILE, decltype([](FILE *f) { fclose(f); })>;

[[nodiscard]] FilePtr fopen(const std::filesystem::path &path,
                            std::string_view mode);
```

https://godbolt.org/z/rb7TvhGc9

Easy or Hard to use wrong?

How do we fix this?

- Maybe it's possible to make this some sort of compile-time type checked set?
- Maybe some bit flags would work?
- Maybe this is truly open ended and OS dependent and "stringly typed" is the only option?

Fuzz Your Interfaces

Fuzz Your Interfaces

fuzzer - a tool that tests your API against a set of “random” inputs.

- Should be run with something like address/undefined sanitizers enabled
- Uses your API in ways that you never would
- Can be used with any API with creativity
- Helps discover patterns of misuse internal to your API

Final Summary

- Try to use your API incorrectly

- Try to use your API incorrectly
- Use better naming

- Try to use your API incorrectly
- Use better naming
- Use `[[nodiscard]]` (with reasons) liberally

- Try to use your API incorrectly
- Use better naming
- Use `[[nodiscard]]` (with reasons) liberally
- Never return a raw pointer

- Try to use your API incorrectly
- Use better naming
- Use `[[nodiscard]]` (with reasons) liberally
- Never return a raw pointer
- Use `noexcept` to help indicate the type of error handling

- Try to use your API incorrectly
- Use better naming
- Use `[[nodiscard]]` (with reasons) liberally
- Never return a raw pointer
- Use `noexcept` to help indicate the type of error handling
- Provide consistent, impossible to ignore, in-band error handling

- Try to use your API incorrectly
- Use better naming
- Use `[[nodiscard]]` (with reasons) liberally
- Never return a raw pointer
- Use `noexcept` to help indicate the type of error handling
- Provide consistent, impossible to ignore, in-band error handling
- Use stronger types and avoid default conversions

- Try to use your API incorrectly
- Use better naming
- Use `[[nodiscard]]` (with reasons) liberally
- Never return a raw pointer
- Use `noexcept` to help indicate the type of error handling
- Provide consistent, impossible to ignore, in-band error handling
- Use stronger types and avoid default conversions
- (Sparingly) delete problematic overloads / prevent conversions

- Try to use your API incorrectly
- Use better naming
- Use `[[nodiscard]]` (with reasons) liberally
- Never return a raw pointer
- Use `noexcept` to help indicate the type of error handling
- Provide consistent, impossible to ignore, in-band error handling
- Use stronger types and avoid default conversions
- (Sparingly) delete problematic overloads / prevent conversions
- Avoid passing pointers (only to be used for single/optional objects)

- Try to use your API incorrectly
- Use better naming
- Use `[[nodiscard]]` (with reasons) liberally
- Never return a raw pointer
- Use `noexcept` to help indicate the type of error handling
- Provide consistent, impossible to ignore, in-band error handling
- Use stronger types and avoid default conversions
- (Sparingly) delete problematic overloads / prevent conversions
- Avoid passing pointers (only to be used for single/optional objects)
- Avoid passing smart pointers

- Try to use your API incorrectly
- Use better naming
- Use `[[nodiscard]]` (with reasons) liberally
- Never return a raw pointer
- Use `noexcept` to help indicate the type of error handling
- Provide consistent, impossible to ignore, in-band error handling
- Use stronger types and avoid default conversions
- (Sparingly) delete problematic overloads / prevent conversions
- Avoid passing pointers (only to be used for single/optional objects)
- Avoid passing smart pointers
- Limit your API as much as possible

- Try to use your API incorrectly
- Use better naming
- Use `[[nodiscard]]` (with reasons) liberally
- Never return a raw pointer
- Use `noexcept` to help indicate the type of error handling
- Provide consistent, impossible to ignore, in-band error handling
- Use stronger types and avoid default conversions
- (Sparingly) delete problematic overloads / prevent conversions
- Avoid passing pointers (only to be used for single/optional objects)
- Avoid passing smart pointers
- Limit your API as much as possible
- Fuzz your API

Make Your API Hard To Use Wrong

Oh, and Enable for `constexpr` Unless You Have a Really Good Reason Not To

Jason Turner

- Host of C++ Weekly https://www.youtube.com/c/lefticus1
- Author
  - C++ Best Practices
  - OpCode, Copy and Reference, Object Lifetime Puzzlers
  - https://amzn.to/3xWh8Ox
  - https://leanpub.com/u/jason_turner
- Developer
  - https://cppbestpractices.com
  - https://github.com/cpp-best-practices
- Microsoft MVP for C++ 2015-present

Jason Turner

Independent and available for training and code reviews

- https://articles.emptycrate.com/idocpp