Applied Best Practices

Jason Turner

- First used C++ in 1996
- Co-host of CppCast http://cppcast.com
- Host of C++ Weekly https://www.youtube.com/c/JasonTurner-lefticus
- Co-creator of ChaiScript http://chaiscript.com
- Curator of http://cppbestpractices.com
- Microsoft MVP for C++ 2015-present

Jason Turner

Independent and available for training or contracting

- http://articles.emptycrate.com/idocpp
- http://articles.emptycrate.com/training.html

About my Talks

- Move to the front!
- Please interrupt and ask questions
- This is approximately what my training looks like

Upcoming Events

- CppCon 2018 Training Post Conference - “C++ Best Practices” - 2 Days
- C++ On Sea 2019 Workshop Post Conference - “Applied `constexpr`” - 1 Day

Upcoming Events

- Special Training Event in the works
  - Matt Godbolt, Charley Bay and Jason Turner together for 3 days
  - Summer 2019
  - Denver Area
  - Expect a focus on C++20, error handling and performance
  - 3 very different perspectives and styles of teaching should keep things interesting!
  - Check out https://coloradoplusplus.info for future updates about this class and other upcoming events in Colorado

Why Applied Best Practices?

Best Practices and Me

- 2007 - 2011 - Series of blog posts on proper usage of the language (RAII, etc).
- C++Now, 2015 - Thinking Portable
- May 2015 - https://cppbestpractices.com
- March 2016 - Learning C++ Best Practices - O'Reilly Video Series
- C++Now, CppCon 2016 - Practical Performance Practices
- July 2017 - C++ Best Practices Class - First time taught, many times since then
- CppCon, Meeting C++ 2017 - Practical C++17
- Pacific++ 2017 - Rethinking Exceptions
- Meeting C++ 2017 - Practical `constexpr`

Best Practices and Me

- Most of my classes and material has been based on my experience with ChaiScript.
- ChaiScript was initially developed with Boost + C++03 in 2009 with cross-compiler, cross-OS compatibility required.
- We've removed Boost and went from C++03 -> C++11 -> C++14 -> C++17, trying to follow best practices along the way.
- The effort was worth it, the code is more maintainable and considerably faster.
- But uniformly applying best practices in an existing code base is very difficult

Best Practices and Me

Early design decisions, such as relying on `shared_ptr` for reference counted objects cannot be moved away from without changing the nature of the system (in this case a scripting language).

I needed a new project.

One that I could apply all these best practices on from the start without worrying about breaking backward compatibility.

Picking a Project To Work On

I often get asked how to learn C++, and my answer is to work on a real project.

Specifically:

1. Something that sounds easy.
2. Something that interests you.

Projects that sound easy almost never are. This is good, we get interested in the project before we realize we are in over our heads.

My Project

I decided to create a simple ARM emulator. I've never written an emulator or had to deal with some of the issues that game development needs to deal with (framerate, etc).

Who watches C++ Weekly?

Did you see the episode about my ARM emulator?

Strong Typing

Strong Typing

I wanted strongly typed integers for this CPU emulator, for more expressive code. For our purposes these values are immutable.

```
1  /// Create a set of strongly typed int32_t
2  struct Op : Strongly_Typed<std::uint32_t, Op> { };
3  struct ALU_Op : Strongly_Typed<std::uint32_t, Op> { };
4  // ... etc
5
6  process(const Op ins);
7  process(const ALU_Op ins);
8  // ... etc
```

Accessors

```
1  template<typename Type, typename CTRP>
2  struct Strongly_Typed {
3    const Type m_data; /// do we like this?
4  };
```

Accessors

```
1  template<typename Type, typename CTRP>
2  struct Strongly_Typed {
3    const Type m_data; /// probably not, reduces usability too much
4  };
```

Accessors

```
1  template<typename Type, typename CTRP>
2  struct Strongly_Typed {
3    /// Need an accessor now
4  protected:
5    Type m_data;
6  };
```

Accessors

Do we get the `m_value` by value or by reference?

```
1  template<typename Type, typename CTRP>
2  struct Strongly_Typed {
3    auto value() { return m_value; } /// Value
4
5    const auto &value() { return m_value; } /// or ref?
6  protected:
7    Type m_value;
8  };
```

Accessors

```
1  template<typename Type, typename CTRP>
2  struct Strongly_Typed {
3    /// Value, intended for small trivial types
4    auto value() { return m_value; }
5
6  protected:
7    Type m_value;
8  };
```

Accessors

```
1  template<typename Type, typename CTRP>
2  struct Strongly_Typed {
3    /// enforce this expectation of trivial types
4    static_assert(std::is_trivial_v<Type>);
5    auto value() { return m_value; }
6
7  protected:
8    Type m_value;
9  };
```

Accessors

```
1  template<typename Type, typename CTRP>
2  struct Strongly_Typed {
3    //  enforce this expectation of trivial types
4    static_assert(std::is_trivial_v<Type>);
5    auto value() { return m_value; }          /// should this be `const`?
6
7  protected:
8    Type m_value;
9  };
```

Accessors

```
1  template<typename Type, typename CTRP>
2  struct Strongly_Typed {
3    //  enforce this expectation of trivial types
4    static_assert(std::is_trivial_v<Type>);
5    auto value() const { return m_value; }  ///
6
7  protected:
8    Type m_value;
9  };
```

Accessors

```
1  template<typename Type, typename CTRP>
2  struct Strongly_Typed {
3    //  enforce this expectation of trivial types
4    static_assert(std::is_trivial_v<Type>);
5    auto value() const { return m_value; }  /// can it throw an exception?
6
7  protected:
8    Type m_value;
9  };
```

Accessors

```
1  template<typename Type, typename CTRP>
2  struct Strongly_Typed {
3    //  enforce this expectation of trivial types
4    static_assert(std::is_trivial_v<Type>);
5    auto value() const noexcept { return m_value; }
6
7  protected:
8    Type m_value;
9  };
```

Accessors

```
1  template<typename Type, typename CTRP>
2  struct Strongly_Typed {
3    //  enforce this expectation of trivial types
4    static_assert(std::is_trivial_v<Type>);
5    auto value() const noexcept { return m_value; } /// can it be constexpr?
6
7  protected:
8    Type m_value;
9  };
```

Accessors

```
1  template<typename Type, typename CTRP>
2  struct Strongly_Typed {
3    //  enforce this expectation of trivial types
4    static_assert(std::is_trivial_v<Type>);
5    constexpr auto value() const noexcept { return m_value; }
6
7  protected:
8    Type m_value;
9  };
```

Accessors

```
1  template<typename Type, typename CTRP>
2  struct Strongly_Typed {
3    //  enforce this expectation of trivial types
4    static_assert(std::is_trivial_v<Type>);
5    /// is it an error to call this function and not use the return value?
6    constexpr auto value() const noexcept { return m_value; }
7
8  protected:
9    Type m_value;
10 };
```

Accessors

```
1  template<typename Type, typename CTRP>
2  struct Strongly_Typed {
3    //  enforce this expectation of trivial types
4    static_assert(std::is_trivial_v<Type>);
5
6    [[nodiscard]] constexpr auto value() const noexcept { return m_value; }
7
8  protected:
9    Type m_value;
10 };
```

Accessors

```
1  template<typename Type, typename CTRP>
2  struct Strongly_Typed {
3    //  enforce this expectation of trivial types
4    static_assert(std::is_trivial_v<Type>);
5    /// are we OK with this?
6    [[nodiscard]] constexpr auto value() const noexcept { return m_value; }
7
8  protected:
9    Type m_value;
10 };
```

On Return Types

Trailing Return Types

Which do we prefer?

```
1  /// A
2  [[nodiscard]] constexpr auto value() const noexcept {return m_value;}
```

```
1  /// B
2  [[nodiscard]] constexpr auto value() const noexcept ->Type{return m_value;}
```

```
1  /// C
2  [[nodiscard]] constexpr Type value() const noexcept {return m_value;}
```

Trailing Return Types

What if the types are longer and full `auto` isn't practical?

```
1  /// A
2  [[nodiscard]] constexpr auto value() const noexcept->Type;
3  [[nodiscard]] constexpr auto op() const noexcept->std::pair<bool,uint32_t>;
4  [[nodiscard]] constexpr auto type() const noexcept->Op_Type;
```

```
1  /// B
2  [[nodiscard]] constexpr Type value() const noexcept;
3  [[nodiscard]] constexpr std::pair<bool, uint32_t> op() const noexcept;
4  [[nodiscard]] constexpr Op_Type type() const noexcept;
```

Trailing Return Types

Others I have talked to have preferred the trailing return types because the function name gets lost in the rest of the keywords when all the attributes are used.

The Resulting Code

Code Like This

```
1  struct System {
2    template<std::size_t Size>
3    constexpr System(const std::array<std::uint8_t, Size> &memory) noexcept
4    {
5      static_assert(Size <= RAM_Size);
6      for (std::size_t loc = 0; loc < Size; ++loc) {
7        // cast is safe - we verified this statically
8        write_byte(static_cast<std::uint32_t>(loc), memory[loc]);
9      }
10     i_cache.fill_cache(*this);
11   }
12
13   [[nodiscard]] constexpr bool ops_remaining() const noexcept {/* */}
14   [[nodiscard]] constexpr auto get(const uint32_t PC) noexcept ->Op{/* */}
15
16   constexpr void setup_run(const std::uint32_t loc) noexcept
17   {
18     registers[14] = RAM_Size - 4;
19     PC()          = loc + 4;
20   }
21   // ....
22 };
```

constexpr

What cannot be `constexpr` in our emulator?

Nothing!

What are the downsides of `constexpr`?

Downsides of `constexpr`

Everything must be in a header.

Fortunately `constexpr` doesn't (necessarily) mean slow compilation times. Slow compilation (generally speaking) comes from

- Very large symbol tables
- Both long symbol names and many symbol names contribute to this

Upsides of `constexpr`

Following the best practices we ended up with code like this possible:

```
1  // if the code compiles, the tests have succeeded
2  TEST_CASE("Test arbitrary movs")
3  {
4    // 0: e3a000e9    mov r0, #233    ; 0xe9
5    // 4: e3a0100c    mov r1, #12
6
7    constexpr auto system = run(0xe9,0x00,0xa0,0xe3,0x0c,0x10,0xa0,0xe3);
8    REQUIRE(static_test<system.registers[0] == 233>());
9    REQUIRE(static_test<system.registers[1] == 12>());
10 }
```

That is: full `constexpr` CPU emulation.

Which means we know provably that we can execute any arbitrary code at compile time. Which at this point really shouldn't be surprising with the work that Hana has done on her `constexpr` regex.

How `constexpr` Helps

What value is returned from main?

```
1  auto shift(int val, int distance)
2  {
3    return val << distance;
4  }
5
6  int main()
7  {
8    auto result = shift(1, 32);
9    return result;
10 }
```

How `constexpr` Helps

Unknown, you cannot shift >= number of bits in the value without invoking undefined behavior.

```
1  auto shift(int val, int distance)
2  {
3    return val << distance;
4  }
5
6  int main()
7  {
8    auto result = shift(1, 32);
9    return result;
10 }
```

How `constexpr` Helps

Now what happens?

```
1  constexpr auto shift(int val, int distance)
2  {
3    return val << distance;
4  }
5
6  int main()
7  {
8    constexpr auto result = shift(1, 32);
9    return result;
10 }
```

How `constexpr` Helps

Cannot compile! By utilizing `constexpr` we can catch extra classes of undefined behavior that warnings cannot catch.

```
1  constexpr auto shift(int val, int distance)
2  {
3    return val << distance;
4  }
5
6  int main()
7  {
8    constexpr auto result = shift(1, 32);
9    return result;
10 }
```

`constexpr` And Undefined Behavior

`constexpr` And Undefined Behavior

- Different compilers have different levels of conformance for not allowing UB
- Portability and testing against multiple compilers is very important (re: Thinking Portable)

`constexpr` And Undefined Behavior

By enabling my emulator for full `constexpr` support and having `constexpr` tests I get an extra level of guarantee that I'm not invoking UB.

C++'s Defaults

C++’s Defaults

What do we think when we see this?

```
1 | [[nodiscard]] constexpr auto value() const noexcept ->Type{return m_value;}
```

Are the defaults wrong?

C++'s Defaults

What if all the defaults were reversed?

```
1 | [[nodiscard]] constexpr auto value() const noexcept ->Type{return m_value;}
```

Becomes

```
1 | auto value() -> Type { return m_value; }
```

And we had different keywords like

```
1 | [[discardable]] auto change_things() mutable noexcept(false) -> Type;
```

C++'s Defaults

Of course we cannot do this today, but this looks kind of like something else in C++....

```
1 // this lambda cannot mutate its data and is constexpr by default
2 // and the return type is specified only once, `-> Type`
3 auto value = [this]() [[nodiscard]] noexcept -> Type{/*return something*/};
```

Lessons Learned In Code

Lessons Learned

- It takes discipline to remember `noexcept`, `[[nodiscard]]`
- If you assume `constexpr`, it's easy to stay in `constexpr` world
- You must have `constexpr` tests to make sure your code actually works in a `constexpr` context
- `constexpr` catches undefined behavior
- clang-format is almost a necessity

Lessons Learned - `clang-format`

- It takes discipline. You need the tools to keep you in line.
- I agree with Tony (Post Modern C++), that too strict a style reduces expressiveness
- However, it's too easy to fall into unformatted "get it done" code that clang-format can easily fix up.
- This also makes accepting patches from new people easier, just ask them to run clang-format on the PR.

Lessons Learned - The Unknowns

- It’s very easy to get undisciplined when doing work you don’t understand yet, such as using a new library

Lessons Learned - Compile Time

- It’s too easy to get into the habit of putting everything in header files
- Still put code that isn’t `constexpr` in `.cpp` files
  - Code that relies on external libraries (this also helps keep your experimental undisciplined code separate)
  - Code that needs dynamic memory

I Keep Mentioning `constexpr`

`constexpr` Is Not The Point of This Talk

… but the subset of C++ that works in `constexpr` context is the language many people say they want.

- No (or minimal) undefined behavior
- No exception handling
- No dynamic allocation (that might be changing)
- Types tend to be `trivial` or at least `trivially_destructible`
- Move semantics are mostly irrelevant when objects are `trivially_copyable`

Build System

Build System Is Critical

- I already knew this, but was still caught by surprise on one thing
- During prototyping I compiled with no warnings. I set up my build system and had many warnings. What happened?!

There Are Warnings

My prototyping command line looks like this:

```
1 | g++ -std=c++17 -Wall -Wextra -Wshadow -Wpedantic test.cpp
```

Looks pretty good?

And Then There Are Warnings

```
-Wall
-Wextra # reasonable and standard
-Wshadow # warn the user if a variable declaration shadows one from a
         # parent context
-Wnon-virtual-dtor # warn the user if a class with virtual functions has a
                   # non-virtual destructor.
-Wold-style-cast # warn for c-style casts
-Wcast-align # warn for potential performance problem casts
-Wunused # warn on anything being unused
-Woverloaded-virtual # warn if you overload (not override) a virtual func
-Wpedantic # warn if non-standard C++ is used
-Wconversion # warn on type conversions that may lose data
-Wsign-conversion # warn on sign conversions
-Wnull-dereference # warn if a null dereference is detected
-Wdouble-promotion # warn if float is implicit promoted to double
-Wformat=2 # warn on security issues around functions that format output
           # (ie printf)
-Wduplicated-cond # warn if if / else chain has duplicated conditions
-Wduplicated-branches # warn if if / else branches have duplicated code
-Wlogical-op # warn about logical operations being used where bitwise were
              # probably wanted
-Wuseless-cast # warn if you perform a cast to the same type
```

And Then There Are Warnings

```
-Wall
-Wextra # reasonable and standard
-Wshadow # warn the user if a variable declaration shadows one from a
         # parent context
-Wnon-virtual-dtor # warn the user if a class with virtual functions has a
                   # non-virtual destructor.
-Wold-style-cast # warn for c-style casts
-Wcast-align # warn for potential performance problem casts
-Wunused # warn on anything being unused
-Woverloaded-virtual # warn if you overload (not override) a virtual func
-Wpedantic # warn if non-standard C++ is used
-Wconversion # warn on type conversions that may lose data
-Wsign-conversion # warn on sign conversions
-Wnull-dereference # warn if a null dereference is detected
-Wdouble-promotion # warn if float is implicit promoted to double
-Wformat=2 # warn on security issues around functions that format output
           # (ie printf)
-Wduplicated-cond # warn if if / else chain has duplicated conditions
-Wduplicated-branches # warn if if / else branches have duplicated code
-Wlogical-op # warn about logical operations being used where bitwise were
              # probably wanted
-Wuseless-cast # warn if you perform a cast to the same type
-Wlifetime # ///
```

And Then There Are Warnings

Even on a trivially sized prototype project, the warnings and type conversions can get out of hand if you don’t start with them enabled.

On The Topic Of Warnings

I did a Twitter query on what feature would you remove from C++ if you could.

Anyone want to guess what the main answer was?

Implicit conversions.

Anyone want to guess what warning I have the hardest time convincing people to turn on?

`-Wconversions`

Warning On Implicit Conversions

I’ve only had one case where this was a very annoying to address issue for me.

```
1 -     PC() += offset + 4;
2 +     PC() += static_cast<std::uint32_t>(offset + 4); // rely on 2's comp
```

Warnings On `switch`es

Warnings On `switch`es

What warning might we get?

```
1  enum class Types { Option1, Option2 };
2
3  auto process(Types t)
4  {
5    switch (t) {
6      case Types::Option1: return handle(T1{t});
7      case Types::Option2: return handle(T2{t});
8    }
9  }
```

Warnings On `switch`es

“Not all paths return a value” - what do we do about it?

```
1  enum class Types { Option1, Option2 };
2
3  auto process(Types t)
4  {
5    switch (t) {
6      case Types::Option1: return handle(T1{t});
7      case Types::Option2: return handle(T2{t});
8    }
9  }
```

Warnings On `switch`es

Let's look at code generated:

```
1  enum class Types { Option1, Option2 };
2
3  struct T1{ Types v; };      struct T2{ Types v; };
4  int handle(T1);             int handle(T2);
5
6  auto process(Types t) {
7    switch (t) {
8      case Types::Option1: return handle(T1{t});
9      case Types::Option2: return handle(T2{t});
10   }
11 }
```

Warnings On `switch`es

What if we add another option?

```
1  enum class Types { Option1, Option2, Option3 };
2
3  struct T1{ Types v; }; struct T2{ Types v; };
4  int handle(T1);        int handle(T2);
5
6  auto process(Types t) {
7    switch (t) {
8      case Types::Option1: return handle(T1{t});
9      case Types::Option2: return handle(T2{t});
10   }
11 }
```

Warnings On `switch`es

What if we pass an invalid option?

```
1  enum class Types { Option1, Option2 };
2
3  struct T1{ Types v; }; struct T2{ Types v; };
4  int handle(T1);        int handle(T2);
5
6  auto process(Types t) {
7    switch (t) {
8      case Types::Option1: return handle(T1{t});
9      case Types::Option2: return handle(T2{t});
10   }
11 }
12
13 int main() {
14   /// Before we continue, is this illegal in any way?
15   process(static_cast<Types>(3)); // See GCC also
16 }
```

Valid Values For `enum` Types:

[dcl.enum]

> *For an enumeration whose underlying type is fixed, the values of the enumeration are* ***the values of the underlying type****.*

What is the underlying type of this enum?

```
1 | enum class Types { Option1, Option2 };
```

> *For a scoped enumeration type, the underlying type is* `int` *if it is not explicitly specified.*

Warnings On `switch`es

What if we guard against unexpected fall-throughs?

```
1  #include <cassert>
2
3  enum class Types { Option1, Option2 };
4
5  struct T1{ Types v; }; struct T2{ Types v; };
6  int handle(T1);        int handle(T2);
7
8  auto process(Types t) {
9    switch (t) {
10     case Types::Option1: return handle(T1{t});
11     case Types::Option2: return handle(T2{t});
12   }
13   assert(!"Cannot reach here!"); /// the mysterious reappearing warning?!
14 }
```

Warnings On `switch`es

Or this?

```
1  #include <cstdlib>
2
3  enum class Types { Option1, Option2 };
4
5  struct T1{ Types v; }; struct T2{ Types v; };
6  int handle(T1);        int handle(T2);
7
8  auto process(Types t) {
9    switch (t) {
10     case Types::Option1: return handle(T1{t});
11     case Types::Option2: return handle(T2{t});
12   }
13   abort();
14 }
```

Warnings On `switch`es

Or throw?

```
1  #include <stdexcept>
2
3  enum class Types { Option1, Option2 };
4
5  struct T1{ Types v; }; struct T2{ Types v; };
6  int handle(T1);        int handle(T2);
7
8  auto process(Types t) {
9    switch (t) {
10     case Types::Option1: return handle(T1{t});
11     case Types::Option2: return handle(T2{t});
12   }
13   throw std::runtime_error("Unhandled Opcode");
14 }
```

Warnings On `switch`es

But we are in `constexpr` land and want to be able to gracefully handle reporting of CPU error states…

```
1  #include <stdexcept>
2
3  enum class Types { Option1, Option2 };
4
5  struct T1{ Types v; }; struct T2{ Types v; };
6  int handle(T1);        int handle(T2);
7
8  auto process(Types t) {
9    switch (t) {
10     case Types::Option1: return handle(T1{t});
11     case Types::Option2: return handle(T2{t});
12   }
13   unhandled_instruction(t); /// Flag for future code
14   return {}; /// valid option for this code?
15 }
```

End Subtle Rant On Warnings, Back To Build System

CMake Is Used

- With a personal ~10 years of bad and outdated CMake technique it's hard to get up to date, but worth it.
- `cmake-format` is used to keep the code clean and formatted, as much as possible.

Package Management

Package Management

We should use the tools available, and for Modern C++, this includes package managers.

My project relies on:

- SFML
- rang
- Catch2

And I am also looking at

- spdlog
- {fmt}

Package Managers

I evaluated (~Aug 18, 2018)

- Buckaroo
- build2 + cppget
- Conan (Center)
- Conan
- C++ Archive Network
- Hunter
- qpm
- vcpkg

	SFML 2.5.0	Catch2 2.3.0	rang 3.1.0	{fmt} 5.1.0	spdlog 1.1.0
Buckaroo	2.4.2	-	2.0.0	3.0.1	0.13.0
cppget	-	-	-	-	-
Conan Center	-	2.3.0	-	5.1.0	1.1.0
Conan	2.5.0	2.3.0	3.1.0	5.1.0	1.1.0
CPPAN	2.5.0	2.2.3	2.0.0	5.1.0	1.0.0
Hunter	-	2.2.2	-	4.1.0	0.16.3
qpm	-	-	-	-	-
vcpkg	2.5.0 *	2.3.0	-	5.1.0	1.0.0

Reinventing the Wheel

So I had this conversation with my cousin, who recently worked for Mozilla on Rust

me: So I wrote an ELF parser for my project.

him: (5 minutes later) look at what I just did with the ELF crate for Rust!

me: I didn’t even *consider* looking for an existing ELF parser for C++

Reinventing the Wheel

We probably need a change of mindset in the C++ community, to think to look for a library that we can use first, instead of just assuming we need to make our own.

Thoughts on Package Management

I've only used package manager in Ruby and Python in situations where licenses and dependencies did not matter much:

- Package managers make it very easy, almost *too* easy to add dependencies. You still need to consider portability and licenses.
- It would be awesome if package managers added an automatic license compatibility checker.
- From Patricia Aas - Make It Fixable: Preparing for Security Vulnerability Reports - CppCon 2018 - Our package managers should warn / error if we install a package with a known vulnerability

Thinking Portable

Thinking Portable

- As mentioned in the previous talk by this name, virtually every C++ program I've ever written has needed to run on multiple platforms
- I prefer developing on Linux, but want to reach a wider market with Windows (and maybe MacOS)

Thinking Portable

- Rely on the std library, `thread`, `filesystem`, and `regex` are huge helps
- None of those are perfect, but they are all good enough
- Be sure to research each package you depend on for compiler / OS support

The Project

The Project

Who saw my Commodore 64 talk from 2016?

The Project

Rich Code For Tiny Computers

- Each line of C++ carefully crafted
- Fragile to the instructions generated
- Couldn’t handle function calls

But…

- Super fun to work on
- Taught me a lot about compilers and zero cost abstractions

The Project

- I wanted to bring this fun to the others
- Without the limitations
- And as a learning tool for C++

The Goal

Commodore BASIC meets C++ and Compiler Explorer

Demo Time

My “Sounds Simple” Project

Eventually contained

- ARM CPU Emulator (Expected)
- GUI frontend (Expected)
- Basic ELF parser (Unexpected)
- Simple linker (Unexpected)
- Partial stdlib implementation (Unexpected)

Where To From Here?

The concept is to gamify best practices while also being an example of best practices.

Eventually it will have challenges with

Positive points for

- Warnings enabled
- Use of `const`

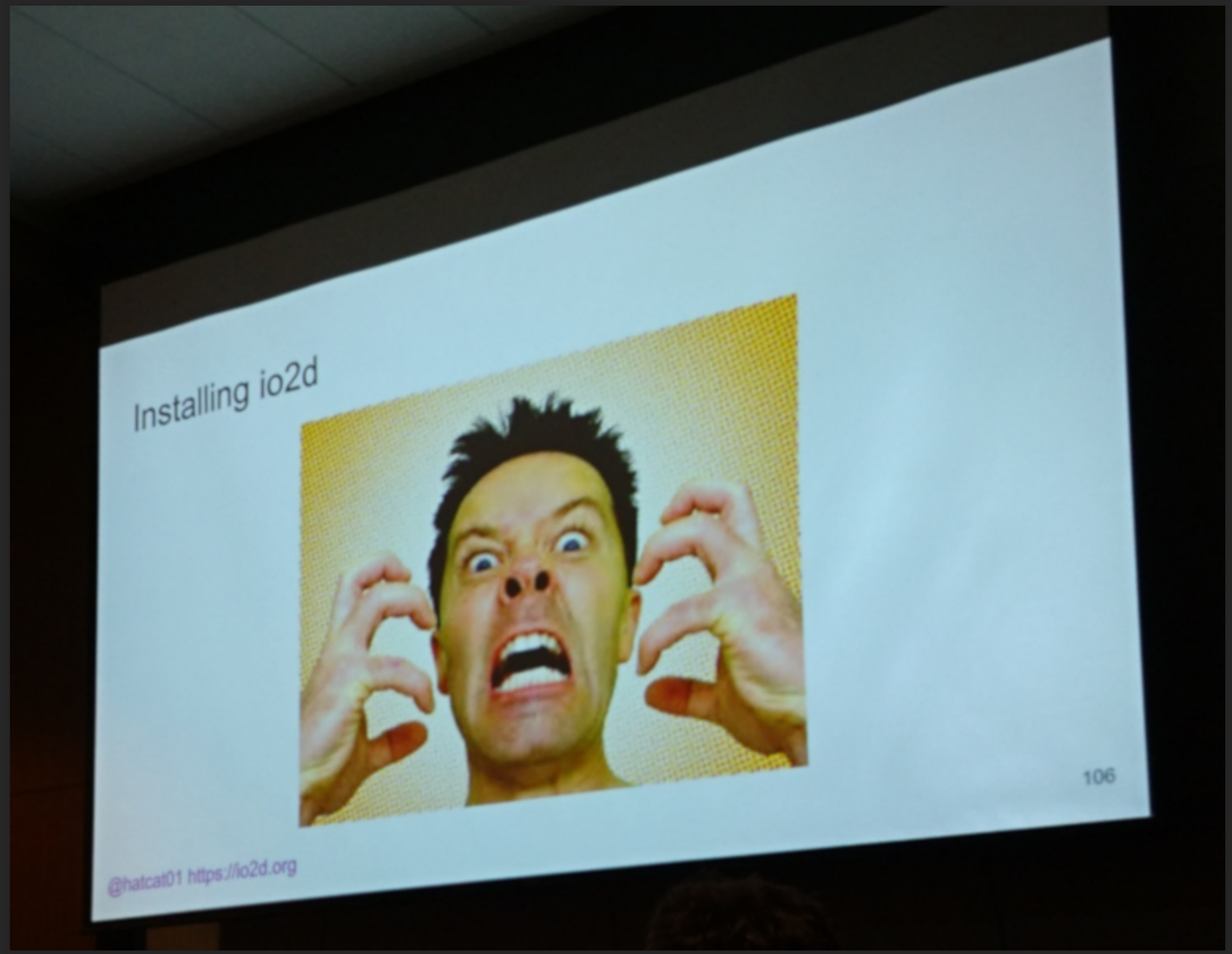
Negative points for

- Dynamic allocations
- CPU instructions executed
- Warnings generated

Limitations

- stdlib support is limited at the moment
- No FPU support (yet)
- No exceptions
- No RTTI
- No static initialization (yet)

On Compiling



(from Guy Davidson, io2d, CppCon2018)

Compiling The C++ Box

Appveyor.yml

```
mkdir build
cd build
pip install --user conan
C:\path\to\conan remote add bincrafters  \
     https://api.bintray.com/conan/bincrafters/public-conan
C:\path\to\conan install .. --build missing
cmake c:\projects\source -G "Visual Studio 15 2017 Win64"
cmake --build . --config "Release"
```

Final Thoughts

Final Thoughts

Is it faster to write simpler code?

No, No, No, No!

- Kate Gregory - CppCon 2018

Final Thoughts

There is still a lot of simplification to do, but currently at ~2500 LOC for:

- ARM Emulator
- ELF Parser
- Simple Linker
- GUI Front End

Final Thoughts

URL: github.com/lefticus/cpp_box

- The project itself should be an example of best practices and easy to read code.
- Continuous Integration and testing is key, and painful to add after the fact. Be sure to start with it from the beginning.
- The pain of setting up CI helps your refine your build story

Jason Turner

- First used C++ in 1996
- Co-host of CppCast http://cppcast.com
- Host of C++ Weekly https://www.youtube.com/c/JasonTurner-lefticus
- Co-creator of ChaiScript http://chaiscript.com
- Curator of http://cppbestpractices.com
- Microsoft MVP for C++ 2015-present

 @lefticus

Copyright Jason Turner @lefticus

Jason Turner

Independent and available for training or contracting

- http://articles.emptycrate.com/idocpp
- http://articles.emptycrate.com/training.html

Upcoming Events

- CppCon 2018 Training Post Conference - “C++ Best Practices” - 2 Days
- C++ On Sea 2019 Workshop Post Conference - “Applied `constexpr`” - 1 Day

Upcoming Events

- Special Training Event in the works
  - Matt Godbolt, Charley Bay and Jason Turner together for 3 days
  - Summer 2019
  - Denver Area
  - Expect a focus on C++20, error handling and performance
  - 3 very different perspectives and styles of teaching should keep things interesting!
  - Check out https://coloradoplusplus.info for future updates about this class and other upcoming events in Colorado