Secrets of C++ Scripting Bindings:

Bridging Compile Time and Run Time

JASON TURNER

Cppcon
The C++ Conference

2024

September 15 - 20

@lefticus

emptycrate.com/idocpp

Secrets of Scripting Bindings for C++

Jason Turner

C++ Weekly

- Weekly videos since March, 2016
- 112k+ subscribers, 445+ weeks straight

https://www.youtube.com/@cppweekly

Jason Turner

- Author
  - C++ Best Practices, C++23 Best Practices
  - OpCode, Copy and Reference, Object Lifetime Puzzlers
  - https://amzn.to/3xWh8Ox
  - https://leanpub.com/u/jason_turner

Jason Turner

- Developer
  - https://cppbestpractices.com
  - https://github.com/lefticus
  - https://github.com/cpp-best-practices
- Microsoft MVP for C++, 2015-present

Jason Turner - Training

https://articles.emptycrate.com/training.html

How to get my training:

1. Have me come to your company on-site for dynamic customized training where you already are - generally the most economical option for groups (DE, NL, RO, CZ, JP, US, PL, SE, …)
2. Come to a conference workshop
   - C++ On Sea (Folkestone, UK, ~July)
   - CppCon (Aurora, CO, US, ~Sept)
   - NDC TechTown (Kongsberg, NO, ~Sept)
   - C++ Under the Sea (Breda, NL, ~Oct)
   - And possibly others

About my Talks

- Avoid sitting in the back
- Please interrupt and ask questions, yell things out, I'll repeat it for the room
- This is approximately how my training days look, as interactive as reasonable

Why This Talk?

Why This Talk?

- ~2006 - I started looking at embedding scripting engine in C++
  - Learned about SWIG
  - Learned that Python is wrong for embedding (Global state), chose Lua
- 2008 - Created SWIG Starter Kit
  - SWIG is great for binding to other languages, but requires a second build step
  - (Side note: this project launched my contracting career)
  - I convinced myself I could do something automagic for C++ users
- 2009 - Started ChaiScript with my cousin Sophia

SWIG

SWIG

What is SWIG?

- Simplified
- Wrapper
- Interface
- Generator

Can either parse interface files or direct header files for C++ libraries

SWIG

Can generate wrapper libraries for:

- C#
- D
- Go
- Guile
- Java
- Javascript
- Lua
- MzScheme/Racket
- OCaml

SWIG

Can generate wrapper libraries for:

- Octave
- Perl
- PHP
- Python
- R
- Ruby
- Scilab
- Tcl

SWIG

I have personally used it for:

- C#
- Java
- Javascript
- Lua
- Python
- Ruby

What About Boost::Python?

What About Boost::Python?

Boost::Python (and similar later tools) are great, but Python was never intended for embedding, presents a crossplatform headache, and shared global state is a problem.

What About Sol2 / Luabind?

What About Sol2 / Luabind?

What About Sol2 / Luabind?

- Lua is mature

What About Sol2 / Luabind?

- Lua is mature
- Lua is designed for embedding

What About Sol2 / Luabind?

- Lua is mature
- Lua is designed for embedding
- If sol2 was a thing when I started ChaiScript, I probably would not have started ChaiScript

Don’t Underestimate Ignorance

ChaiScript

ChaiScript

- Header-only scripting engine designed for embedding in C++

ChaiScript

- Header-only scripting engine designed for embedding in C++
- Automatic function / type deduction

ChaiScript

- Header-only scripting engine designed for embedding in C++
- Automatic function / type deduction
- Native script function <-> C++ Function interaction

ChaiScript

- Header-only scripting engine designed for embedding in C++
- Automatic function / type deduction
- Native script function <-> C++ Function interaction
- Full support for exceptions

ChaiScript

- Header-only scripting engine designed for embedding in C++
- Automatic function / type deduction
- Native script function <-> C++ Function interaction
- Full support for exceptions
- Just Works (TM)

ChaiScript

```
1  #include <chaiscript/chaiscript.hpp>
2
3  std::string helloWorld(const std::string &t_name) {
4    return "Hello " + t_name + "!";
5  }
6
7  int main() {
8    chaiscript::ChaiScript chai;
9    chai.add(chaiscript::fun(&helloWorld), "helloWorld");
10
11   chai.eval(R"(
12     puts(helloWorld("Bob"));
13   )");
14 }
```

https://godbolt.org/z/z33z3qqoo

ChaiScript

ChaiScript had a moderate amount of success, and through its development and via bug reports and contributions from users, I learned about:

ChaiScript

ChaiScript had a moderate amount of success, and through its development and via bug reports and contributions from users, I learned about:

- Template Meta Programming

ChaiScript

ChaiScript had a moderate amount of success, and through its development and via bug reports and contributions from users, I learned about:

- Template Meta Programming
- Lambdas

ChaiScript

ChaiScript had a moderate amount of success, and through its development and via bug reports and contributions from users, I learned about:

- Template Meta Programming
- Lambdas
- `constexpr`

ChaiScript

ChaiScript had a moderate amount of success, and through its development and via bug reports and contributions from users, I learned about:

- Template Meta Programming
- Lambdas
- `constexpr`
- Static Analysis

ChaiScript

ChaiScript had a moderate amount of success, and through its development and via bug reports and contributions from users, I learned about:

- Template Meta Programming
- Lambdas
- `constexpr`
- Static Analysis
- Compiler Warning Options

ChaiScript

ChaiScript had a moderate amount of success, and through its development and via bug reports and contributions from users, I learned about:

- Template Meta Programming
- Lambdas
- `constexpr`
- Static Analysis
- Compiler Warning Options
- Runtime Analysis

ChaiScript

ChaiScript had a moderate amount of success, and through its development and via bug reports and contributions from users, I learned about:

- Template Meta Programming
- Lambdas
- `constexpr`
- Static Analysis
- Compiler Warning Options
- Runtime Analysis
- Sanitizers

ChaiScript

ChaiScript had a moderate amount of success, and through its development and via bug reports and contributions from users, I learned about:

- Template Meta Programming
- Lambdas
- `constexpr`
- Static Analysis
- Compiler Warning Options
- Runtime Analysis
- Sanitizers
- Fuzz Testing

ChaiScript

It is probably the single most important project I've worked on in my career.

Why This Talk?

Why This Talk?

- 2010-2023 - Consulted on C++/Scripting projects

Why This Talk?

- 2010-2023 - Consulted on C++/Scripting projects
- 2016 - C++Now “Why and How to Add Scripting”

Why This Talk?

- 2010-2023 - Consulted on C++/Scripting projects
- 2016 - C++Now “Why and How to Add Scripting”
- 2011-2017 - Ported ChaiScript from Boost → C++11 → C++14 → C++17

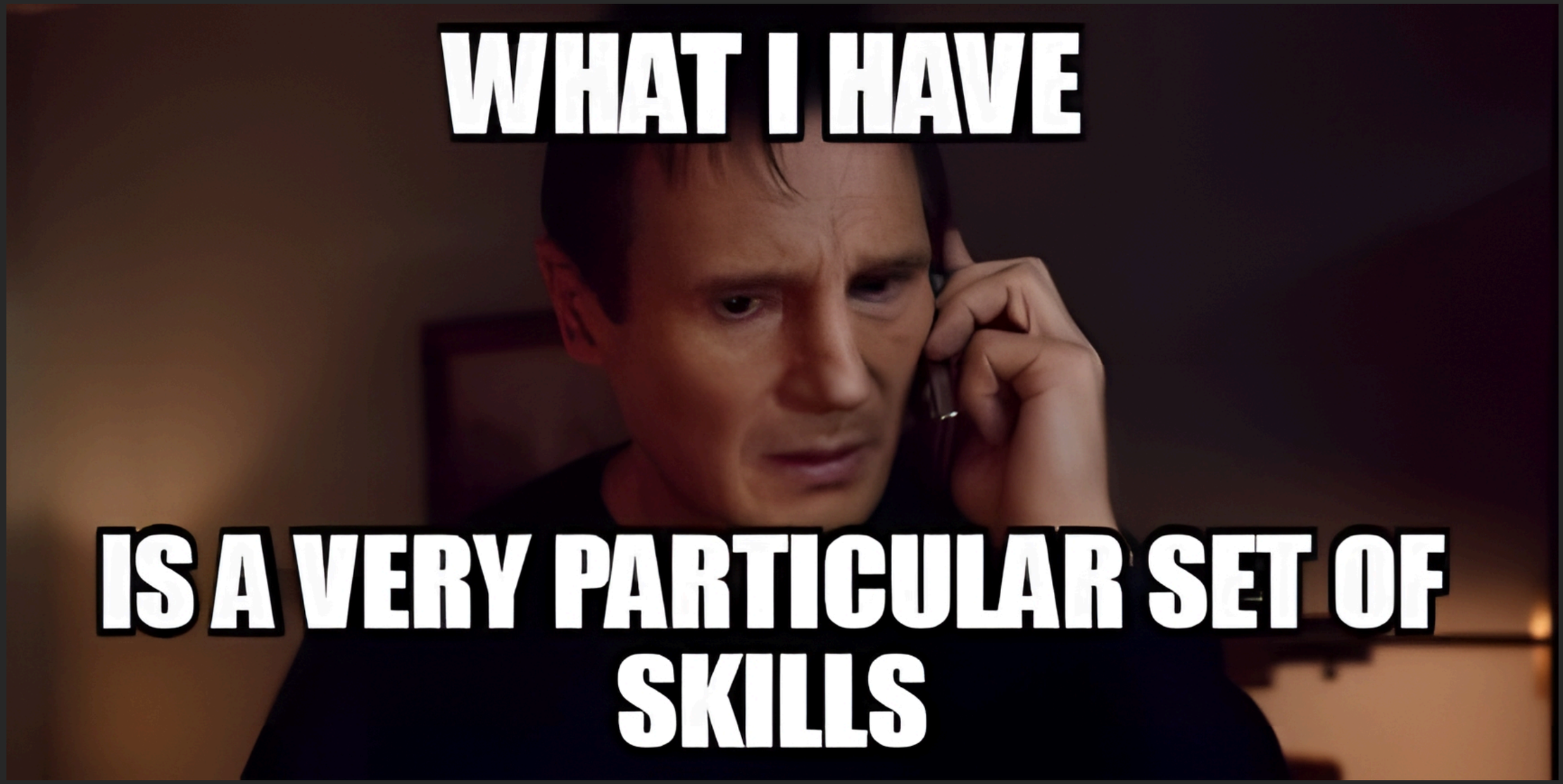
Why This Talk?

- 2010-2023 - Consulted on C++/Scripting projects
- 2016 - C++Now “Why and How to Add Scripting”
- 2011-2017 - Ported ChaiScript from Boost → C++11 → C++14 → C++17
- 2019-2023 - Pondered making a newer `constexpr` friendly embedded scripting engine

Why This Talk?

- 2010-2023 - Consulted on C++/Scripting projects
- 2016 - C++Now “Why and How to Add Scripting”
- 2011-2017 - Ported ChaiScript from Boost → C++11 → C++14 → C++17
- 2019-2023 - Pondered making a newer `constexpr` friendly embedded scripting engine
- 2023 - Created cons_expr, a scheme-inspired embedded scripting engine with no dynamic allocations, no exceptions, and 100% `constexpr` capable

In Other Words



Goals For This Talk

Goals

Goals

1. You'll understand the challenges of bridging compile time ⇄ run time worlds

Goals

1. You'll understand the challenges of bridging compile time ⇄ run time worlds
2. You'll understand one possible solution

Goals

1. You'll understand the challenges of bridging compile time ⇄ run time worlds
2. You'll understand one possible solution
3. You'll be able to pick up from the simple example and build your own simple scripting tool

Code Goals

Code Goals

1. Simple

Code Goals

1. Simple
2. Succinct

Code Goals

1. Simple
2. Succinct
3. Readable

Code Goals

1. Simple
2. Succinct
3. Readable
4. Leverages the standard library

Code Non-Goals

Code Non-Goals

1. We are not worried about performance

Code Non-Goals

1. We are not worried about performance
2. We are not worried about compile-time

Code Non-Goals

1. We are not worried about performance
2. We are not worried about compile-time
3. We are not worried about binary size

The Parts

The Parts

1. Way to register function with engine
2. Way to call function with runtime known values

Registering Functions

Registering Functions

```
1  int add(int, int);
2  int abs(int);
3  void print(int);
4  void print(std::string_view);
5
6  struct ScriptEngine {
7    void add(/**/); /// what here?
8  };
9
10 int main() {
11   ScriptEngine se;
12   se.add(&add, "add");
13   se.add(&abs, "abs");
14   se.add(&print, "print_int");
15   se.add(&print, "print_string");
16 }
```

https://godbolt.org/z/W8M7Wed7j

A Function That Can Match Functions

A Function That Can Match Functions

A Function That Can Match Functions

- Must be a template

A Function That Can Match Functions

- Must be a template
- Uses template pattern matching

A Function That Can Match Functions

- Must be a template
- Uses template pattern matching

```
1 template<typename Ret, typename ... Param>
2 void add(Ret (*func)(Param...)); // can match any free function
```

A Function That Can Match Functions

We’re going to concern ourselves with just free function pointers today, but in total we must deal with:

A Function That Can Match Functions

We’re going to concern ourselves with just free function pointers today, but in total we must deal with:

- Free functions (which include class static member functions)

A Function That Can Match Functions

We’re going to concern ourselves with just free function pointers today, but in total we must deal with:

- Free functions (which include class static member functions)
  - `noexcept(true)` and `noexcept(false)` (2)

A Function That Can Match Functions

We're going to concern ourselves with just free function pointers today, but in total we must deal with:

- Free functions (which include class static member functions)
  - `noexcept(true)` and `noexcept(false)` (2)
- Member functions (which includes?)

A Function That Can Match Functions

We're going to concern ourselves with just free function pointers today, but in total we must deal with:

- Free functions (which include class static member functions)
  - `noexcept(true)` and `noexcept(false)` (2)
- Member functions (which includes?)
  - non-cv qualified, `const`, `volatile`, `const volatile` (4)

A Function That Can Match Functions

We're going to concern ourselves with just free function pointers today, but in total we must deal with:

- Free functions (which include class static member functions)
  - `noexcept(true)` and `noexcept(false)` (2)
- Member functions (which includes?)
  - non-cv qualified, `const`, `volatile`, `const volatile` (4)
  - non-reference qualified, `&`, `&&` (* 3)

A Function That Can Match Functions

We're going to concern ourselves with just free function pointers today, but in total we must deal with:

- Free functions (which include class static member functions)
  - `noexcept(true)` and `noexcept(false)` (2)
- Member functions (which includes?)
  - non-cv qualified, `const`, `volatile`, `const volatile` (4)
  - non-reference qualified, `&`, `&&` (* 3)
  - `noexcept(true)` and `noexcept(false)` (* 2)

A Function That Can Match Functions

We're going to concern ourselves with just free function pointers today, but in total we must deal with:

- Free functions (which include class static member functions)
  - `noexcept(true)` and `noexcept(false)` (2)
- Member functions (which includes?)
  - non-cv qualified, `const`, `volatile`, `const volatile` (4)
  - non-reference qualified, `&`, `&&` (* 3)
  - `noexcept(true)` and `noexcept(false)` (* 2)
- Objects with call operator overloaded (1)

A Function That Can Match Functions

We're going to concern ourselves with just free function pointers today, but in total we must deal with:

- Free functions (which include class static member functions)
  - `noexcept(true)` and `noexcept(false)` (2)
- Member functions (which includes?)
  - non-cv qualified, `const`, `volatile`, `const volatile` (4)
  - non-reference qualified, `&`, `&&` (* 3)
  - `noexcept(true)` and `noexcept(false)` (* 2)
- Objects with call operator overloaded (1)

2 + (4 * 3 * 2) + 1 = 27 if I counted correctly

A Function That Can Match Functions

```
1  template<typename Ret, typename ... Param> void add(Ret (*f)(Param...));
2  template<typename Ret, typename ... Param> void add(Ret (*f)(Param...) noexcept);
3
4  template<typename Ret, typename Class, typename ... Param>
5  void add(Ret (Class::*f)(Param...));
6  template<typename Ret, typename Class, typename ... Param>
7  void add(Ret (Class::*f)(Param...) const);
8  template<typename Ret, typename Class, typename ... Param>
9  void add(Ret (Class::*f)(Param...) volatile);
10 template<typename Ret, typename Class, typename ... Param>
11 void add(Ret (Class::*f)(Param...) const volatile);
12 template<typename Ret, typename Class, typename ... Param>
13 void add(Ret (Class::*f)(Param...) &);
14 template<typename Ret, typename Class, typename ... Param>
15 void add(Ret (Class::*f)(Param...) const &);
16 /// and so many more
17
18 template<typename Func> void add(Func &&func) {
19   add(&Func::operator()); /// fallback that then registers overloaded call operator
20 }
```

https://godbolt.org/z/7fWGsq63P

A Function That Can Match Functions

What about static member functions?

```
1  struct S {
2    static int get_value();
3  };
4
5  int main() {
6    add(&S::get_value);
7  }
```

https://godbolt.org/z/17vE4fYf6

A Function That Can Match Functions

What about static member functions?

```
1  struct S {
2    static int get_value();
3  };
4
5  int main() {
6    add(&S::get_value);
7  }
```

https://godbolt.org/z/17vE4fYf6

They are just free functions.

A Function That Can Match Functions

What about explicit `this`?

```
1  struct S {
2    int get_value(this const S &self);
3  };
4
5  int main() {
6    add(&S::get_value);
7  }
```

https://godbolt.org/z/bv6YdMc8o

A Function That Can Match Functions

What about explicit `this`?

```
1  struct S {
2    int get_value(this const S &self);
3  };
4
5  int main() {
6    add(&S::get_value);
7  }
```

https://godbolt.org/z/bv6YdMc8o

They are just like static members.

A Function That Can Match Functions

How did we do this before C++11's variadic templates?

A Function That Can Match Functions

How did we do this before C++11's variadic templates?

```
1  template<typename Ret>
2  void add(Ret (*f)());
3
4  template<typename Ret, typename P1>
5  void add(Ret (*f)(P1));
6
7  template<typename Ret, typename P1, typename P2>
8  void add(Ret (*f)(P1, P2));
9
10 template<typename Ret, typename P1, typename P2, typename P3>
11 void add(Ret (*f)(P1, P2, P3));
12 /*...*/
```

https://godbolt.org/z/YKoGvzeKa

OR...?

A Function That Can Match Functions

How did we do this before C++11's variadic templates?

```
1  template<typename Ret>
2  void add(Ret (*f)());
3
4  template<typename Ret, typename P1>
5  void add(Ret (*f)(P1));
6
7  template<typename Ret, typename P1, typename P2>
8  void add(Ret (*f)(P1, P2));
9
10 template<typename Ret, typename P1, typename P2, typename P3>
11 void add(Ret (*f)(P1, P2, P3));
12 /*...*/
```

https://godbolt.org/z/YKoGvzeKa

OR...?

`BOOST_PP` and lots of macros (not recommended)

Registering Functions

```
1  int add(int, int);
2  int abs(int);
3  void print(int);
4  void print(std::string_view);
5
6
7  struct ScriptEngine {
8    template<typename Ret, typename ... Param>
9    void add(Ret (*f)(Param...));
10 };
11
12 int main() {
13   ScriptEngine se;
14   se.add(&add, "add");
15 }
```

https://godbolt.org/z/cWsqhaezY

Registering Functions

```
1  int add(int, int);
2  int abs(int);
3  void print(int);
4  void print(std::string_view);
5
6
7  struct ScriptEngine {
8    template<typename Ret, typename ... Param>
9    void add(Ret (*f)(Param...));
10 };
11
12 int main() {
13   ScriptEngine se;
14   se.add(&add, "add");
15 }
```

https://godbolt.org/z/cWsqhaezY

How do we store the functions we want to register?

Registering Functions

```
1  int add(int, int);
2  int abs(int);
3  void print(int);
4  void print(std::string_view);
5
6
7  struct ScriptEngine {
8    template<typename Ret, typename ... Param>
9    void add(Ret (*f)(Param...));
10 };
11
12 int main() {
13   ScriptEngine se;
14   se.add(&add, "add");
15 }
```

https://godbolt.org/z/cWsqhaezY

How do we store the functions we want to register?

We need a generic way of storing any possible type of callable.

Registering Functions

```
1  struct ScriptEngine {
2    std::vector</*...*/> functions; /// ??
3
4    template<typename Ret, typename ... Param>
5    void add(Ret (*f)(Param...));
6  };
```

https://godbolt.org/z/qvdzf9r9T

We need a generic mapping for any type of function.

Some way to store any possible function, and call it later.

A Generic Function Signature

What’s the first thing that comes to mind when you think “generic function?”

A Generic Function Signature

What's the first thing that comes to mind when you think "generic function?"

Template?

A Generic Function Signature

What's the first thing that comes to mind when you think "generic function?"

Template?

- Can you store a function template?

A Generic Function Signature

What's the first thing that comes to mind when you think "generic function?"

Template?

- Can you store a function template?
- Can you get a pointer to a function template?

A Generic Function Signature

What's the first thing that comes to mind when you think "generic function?"

Template?

- Can you store a function template?
- Can you get a pointer to a function template?

For the sake of this conversation a helpful mental model is:

A Generic Function Signature

What's the first thing that comes to mind when you think "generic function?"

Template?

- Can you store a function template?
- Can you get a pointer to a function template?

For the sake of this conversation a helpful mental model is:

- Function templates don't exist

A Generic Function Signature

What's the first thing that comes to mind when you think "generic function?"

Template?

- Can you store a function template?
- Can you get a pointer to a function template?

For the sake of this conversation a helpful mental model is:

- Function templates don't exist
- Instantiations of function templates do

A Generic Function Signature

So we need a concrete function (not a template) that can take any number of parameters.

A Generic Function Signature

So we need a concrete function (not a template) that can take any number of parameters.

```
1 | Type generic_function(std::span<Type>);
```

A Generic Function Signature

So we need a concrete function (not a template) that can take any number of parameters.

```
1 | Type generic_function(std::span<Type>);
```

- What should this placeholder `Type` be?

A Generic Function Signature

So we need a concrete function (not a template) that can take any number of parameters.

```
1 | Type generic_function(std::span<Type>);
```

- What should this placeholder `Type` be?
- It needs to be able to hold all of the types we care about plus one other thing…

A Generic Function Signature

So we need a concrete function (not a template) that can take any number of parameters.

```
1 | Type generic_function(std::span<Type>);
```

- What should this placeholder `Type` be?
- It needs to be able to hold all of the types we care about plus one other thing…

`void`

A Generic Function Signature

So we need a concrete function (not a template) that can take any number of parameters.

```
1 | Type generic_function(std::span<Type>);
```

- What should this placeholder `Type` be?
- It needs to be able to hold all of the types we care about plus one other thing…

`void`

Unfortunately `void` is not a regular type, so we have to deal with it separately.

A Generic Function Signature

Functions we initially want to support:

```cpp
int add(int, int);
int abs(int);
void print(int);
void print(std::string_view);
```

https://godbolt.org/z/MvxThfWKr

Now what should be the replacement for `Type`?

```cpp
Type generic_function(std::span<Type>);
```

There are 2 main choices.

A Generic Function Signature

Functions we initially want to support:

```
1 int add(int, int);
2 int abs(int);
3 void print(int);
4 void print(std::string_view);
```

https://godbolt.org/z/MvxThfWKr

Now what should be the replacement for `Type`?

```
1 Type generic_function(std::span<Type>);
```

There are 2 main choices.

- `std::variant<std::monostate, int, std::string_view>`

A Generic Function Signature

Functions we initially want to support:

```
1 int add(int, int);
2 int abs(int);
3 void print(int);
4 void print(std::string_view);
```

https://godbolt.org/z/MvxThfWKr

Now what should be the replacement for `Type`?

```
1 Type generic_function(std::span<Type>);
```

There are 2 main choices.

- `std::variant<std::monostate, int, std::string_view>`
- `std::any`

A Generic Function Signature - `variant`

```cpp
#include <functional>
#include <span>
#include <variant>
#include <vector>

template<typename ... AllowedTypes>
struct ScriptEngine {
  using ParamType = std::variant<std::monostate, AllowedTypes...>;

  std::vector<std::function<ParamType (std::span<ParamType>)>> functions;

  template<typename Ret, typename ... Param>
  void add(Ret (*f)(Param...));
};
```

https://godbolt.org/z/Yefxxfr7M

- Pro - You get exactly what you ask for
- Con - You must declare ahead of time exactly what types you want to support

A Generic Function Signature - `std::any`

```cpp
#include <any>
#include <functional>
#include <span>
#include <vector>

struct ScriptEngine {
  using ParamType = std::any;

  std::vector<std::function<ParamType (std::span<ParamType>)>> functions;

  template<typename Ret, typename ... Param>
  void add(Ret (*f)(Param...));
};
```

https://godbolt.org/z/qPaWMbnsK

- Pro - Works with literally any type that can be copied or moved, user doesn't have to specify. "It's Magic"
- Con - If objects are bigger than the small object optimization, you'll get a heap allocation inside of the `any`

A Generic Function Signature

- `std::any` - The approach taken by ChaiScript. It's magic and trivially easy for the user
- `std::variant<>` - The approach taken by cons_expr. Requires more foreknowledge of the types wanted

We will be using the `std::any` approach for the rest of this talk, but the principles are similar.

```
 1  #include <any>
 2  #include <functional>
 3  #include <span>
 4  #include <vector>
 5
 6  struct ScriptEngine {
 7    std::vector<std::function<std::any (std::span<std::any>)>> functions;
 8
 9    template<typename Ret, typename ... Param>
10    void add(Ret (*f)(Param...));
11  };
```

https://godbolt.org/z/MsYMeono3

A Generic Function Signature

Questions up to this point?

The `add` Function

```
1  std::vector<std::function<std::any (std::span<std::any>)>> functions;
2
3  template<typename Ret, typename ... Param>
4  void add(Ret (*f)(Param...))
5  {
6    ///
7  }
```

https://godbolt.org/z/zKPaer7sb

We need to write a function that takes a function and returns a new function that's generic but knows how to call the original function…

The `add` Function

```
1  std::vector<std::function<std::any (std::span<std::any>)>> functions;
2
3  template<typename Ret, typename ... Param>
4  void add(Ret (*f)(Param...))
5  {
6    functions.push_back(
7      [f](std::span<std::any>) -> std::any {
8
9      };
10   );
11 }
```

https://godbolt.org/z/rWeEPhKcn

The `add` Function

```cpp
#include <any>
#include <functional>
#include <span>
#include <vector>

std::vector<std::function<std::any (std::span<std::any>)>> functions;

template<typename Ret, typename ... Param>
void add(Ret (*f)(Param...))
{
  functions.push_back(
    [f](std::span<std::any> params) -> std::any { // stored lambda
      // a lambda that knows how to take the span of anys
      // and cast them to the desired types
      const auto invoker = // helper lambda
        [&]<std::size_t... Index>(std::index_sequence<Index...>) {
          /// we need to unpack the parameter types and the indices together
          /// this works because they have the same pack size
          /// replace `any_cast` with your own helper that does
          /// any conversions you want.
          return func(std::any_cast<Param>(params[Index])...);
        };

      return invoker(std::make_index_sequence<sizeof...(Param)>());
    }
  );
}
```

https://godbolt.org/z/Y74jse1Wv

The `add` Function

The `add` Function

- We assume contained type is exactly the same as parameter type

The `add` Function

- We assume contained type is exactly the same as parameter type
- A failed `any_cast` will throw an exception

The `add` Function

- We assume contained type is exactly the same as parameter type
- A failed `any_cast` will throw an exception
- If we're passed the wrong number of parameters, we have UB (span.at doesn't exist until C++26)

The `add` Function

- We assume contained type is exactly the same as parameter type
- A failed `any_cast` will throw an exception
- If we're passed the wrong number of parameters, we have UB (span.at doesn't exist until C++26)
- C++20's explicit lambda template parameters are needed to get the index pack

The `add` Function

- We assume contained type is exactly the same as parameter type
- A failed `any_cast` will throw an exception
- If we're passed the wrong number of parameters, we have UB (span.at doesn't exist until C++26)
- C++20's explicit lambda template parameters are needed to get the index pack
- No conversions, not even to `&` is currently supported

We Need More Context

We Need More Context

We can store a function now in a generic way, and (at least theoretically) can call it, but we also need (at least) arity (number of parameters) and name.

```cpp
#include <any>
#include <functional>
#include <map>
#include <span>
#include <string>

struct Function {
  /// std::function because we have a capture
  std::function<std::any(std::span<std::any>)> callable;
  std::size_t arity;
};

std::map<std::string, Function> functions;

template <typename Ret, typename... Param>
void add(std::string name, Ret (*func)(Param...)) {
  functions.try_emplace(
      std::move(name), /// key
      /// value params
      [func](std::span<std::any> params) -> std::any {
        return [&]<std::size_t... Index>(std::index_sequence<Index...>) {
          return func(std::any_cast<Param>(params[Index])...);
        }(std::make_index_sequence<sizeof...(Param)>()); /// IILE
      },
      sizeof...(Param));
}

int main() {
  add("+", +[](int x, int y) { return x + y; }); /// func *
}
```

https://godbolt.org/z/frnrT7h76

Notes

- We can avoid using type erasure like `std::function` by using function pointers, but then we need to ensure that the lambdas don't capture
- We might prefer C++23's `flat_map`, but it's not `constexpr` capable, if you care about that
- This is probably the most `...` and most TMP examples I've ever used in a conference talk
- Globals used for conciseness - we'll fix this later

Invoking a Function

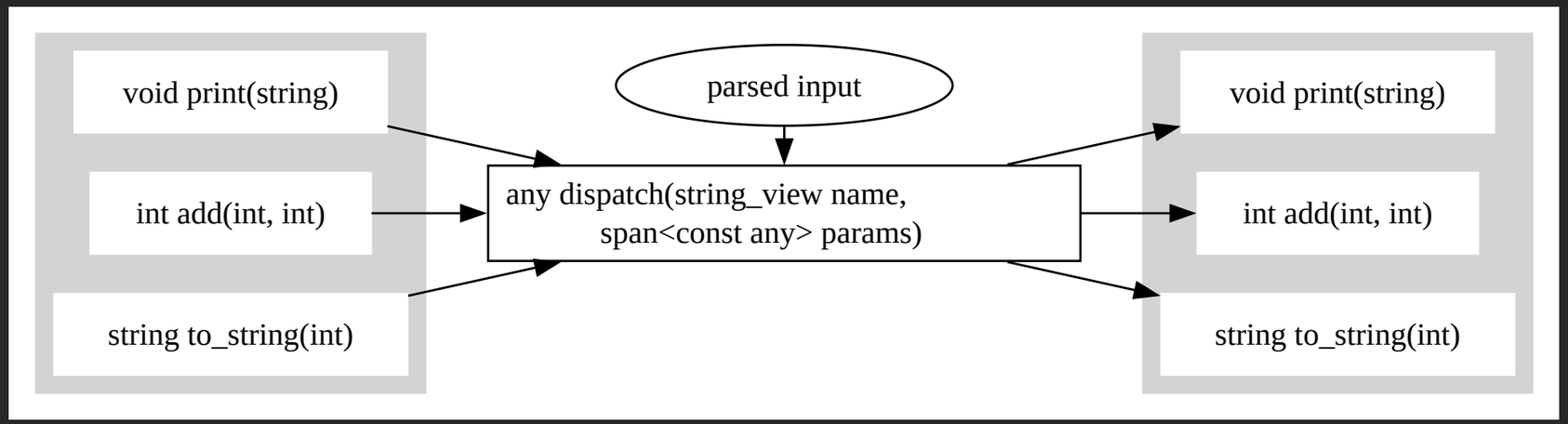
```cpp
#include <any>
#include <functional>
#include <map>
#include <span>
#include <string>
#include <array>

struct Function {
  std::function<std::any(std::span<std::any>)> callable;
  std::size_t arity;
};
std::map<std::string, Function> functions;

template <typename Ret, typename... Param>
void add(std::string name, Ret (*func)(Param...)) {
  functions.try_emplace(
      std::move(name),
      [func](std::span<std::any> params) -> std::any {
        return [&]<std::size_t... Index>(std::index_sequence<Index...>) {
          return func(std::any_cast<Param>(params[Index])...);
        }(std::make_index_sequence<sizeof...(Param)>());
      },
      sizeof...(Param));
}

int main() {
  add("+", +[](int x, int y) { return x + y; });
  std::array<std::any, 2> values{1, 2}; ///
  return std::any_cast<int>(functions.at("+").callable(values)); ///
}
```

https://godbolt.org/z/861hWMddx

I know what you're thinking…

Why jump through all these hoops? What has been gained?

Scripting Funnel



Scripting Integrations

```
1  int main() {
2    add("-", +[](int x, int y) { return x - y; });
3    add("*", +[](int x, int y) { return x * y; });
4    add("to_string", +[](int x) { return std::to_string(x); });
5    add("print", +[](int x) { return std::to_string(x); });
6
7    std::vector<std::any> stack;
8
9    stack.push_back(1);
10   stack.push_back(3);
11   stack.push_back(6);
12
13   eval("*", stack);
14   eval("-", stack);
15   eval("to_string", stack);
16   eval("print", stack);
17
18   return std::any_cast<int>();
19 }
```

https://godbolt.org/z/PafYPYWoa

What have we just executed?

Scripting Integrations

```
1  int main() {
2    add("-", +[](int x, int y) { return x - y; });
3    add("*", +[](int x, int y) { return x * y; });
4    add("to_string", +[](int x) { return std::to_string(x); });
5    add("print", +[](int x) { return std::to_string(x); });
6
7    std::vector<std::any> stack;
8
9    stack.push_back(1);
10   stack.push_back(3);
11   stack.push_back(6);
12
13   eval("*", stack);
14   eval("-", stack);
15   eval("to_string", stack);
16   eval("print", stack);
17
18   return std::any_cast<int>();
19 }
```

https://godbolt.org/z/PafYPYWoa

What have we just executed?

```
1 print(to_string(1 - (3 * 6)))
```

Scripting Integration

Imagine this was your input file:

```
1 | print(to_string(1 - (3 * 6)))
```

Scripting Integration

Imagine this was your input file:

```
print(to_string(1 - (3 * 6)))
```

or

```
(print (to_string (- 1 (* 3 6))))
```

Scripting Integration

Imagine this was your input file:

```
print(to_string(1 - (3 * 6)))
```

or

```
(print (to_string (- 1 (* 3 6))))
```

or (with effectively 0 effort parsing)

```
1
3
6
*
-
to_string
print
```

Inspired Yet?

New Goal

```
#include <string_view>
static constexpr std::string_view script = R"(
Value Was:
1
3
6
*
-
to_string
append
print
)";
```

https://godbolt.org/z/rYWhndvMd

- If the line is empty, it's skipped
- If the value matches a known function, the function is executed
  - appropriate number of values are popped from the stack
  - return value is pushed to the stack
- If the value fully parses as an integer, it's an integer
- Otherwise it's a string literal

Registering All The Functions

Registering All The Functions

```
1  struct Function {
2    std::function<std::any(std::span<std::any>)> callable;
3    std::size_t arity;
4  };
5  std::map<std::string, Function> functions;
6
7  template <typename Ret, typename... Param>
8  void add(std::string name, Ret (*func)(Param...)) {
9    functions.try_emplace(
10       std::move(name),
11       [func](std::span<std::any> params) -> std::any {
12         return [&]<std::size_t... Index>(std::index_sequence<Index...>) {
13           return func(std::any_cast<Param>(params[Index])...);
14         }(std::make_index_sequence<sizeof...(Param)>());
15       },
16       sizeof...(Param));
17 }
18
19 int main() {
20   /// Will this compile?
21   add("print", +[](std::string input) { puts(input.c_str()); });
22 }
```

https://godbolt.org/z/WbM5nG9er

Registering All The Functions

```
1  template <typename Ret, typename... Param>
2  void add(std::string name, Ret (*func)(Param...)) {
3    functions.try_emplace(
4        std::move(name),
5        [func](std::span<std::any> params) -> std::any {
6          return [&]<std::size_t... Index>(std::index_sequence<Index...>) {
7            /// how to handle void return types?
8            return func(std::any_cast<Param>(params[Index])...);
9
10
11
12
13         }(std::make_index_sequence<sizeof...(Param)>());
14       },
15       sizeof...(Param));
16 }
17
18 int main() {
19   add("print", +[](std::string input) { puts(input.c_str()); });
20 }
```

https://godbolt.org/z/1bEPe3c71

Registering All The Functions

```
1  template <typename Ret, typename... Param>
2  void add(std::string name, Ret (*func)(Param...)) {
3    functions.try_emplace(
4        std::move(name),
5        [func](std::span<std::any> params) {
6          return [&]<std::size_t... Index>(std::index_sequence<Index...>)->std::any {
7            if constexpr (!std::is_same_v<void, Ret>)
8              return func(std::any_cast<Param>(params[Index])...);
9            } else  {
10             func(std::any_cast<Param>(params[Index])...);
11             return {};
12           }
13         }(std::make_index_sequence<sizeof...(Param)>());
14       },
15       sizeof...(Param));
16 }
17
18 int main() {
19   add("print", +[](std::string input) { puts(input.c_str()); });
20 }
```

https://godbolt.org/z/ah9Eobrfa

Oh, And You'll Eventually Hit This…

Overloaded C++ Functions

```cpp
template <typename Ret, typename... Param>
void add(std::string name, Ret (*func)(Param...)) {
  functions.try_emplace(
      std::move(name),
      [func](std::span<std::any> params) {
        return [&]<std::size_t... Index>(std::index_sequence<Index...>)->std::any {
          if constexpr (!std::is_same_v<void, Ret>)
            return func(std::any_cast<Param>(params[Index])...);
          } else {
            func(std::any_cast<Param>(params[Index])...);
            return {};
          }
        }(std::make_index_sequence<sizeof...(Param)>());
      },
      sizeof...(Param));
}

void print(int);
void print(std::string);

int main() {
  add("print", &print); // Works?
}
```

https://godbolt.org/z/aKsq6zbGa

Overloaded C++ Functions

```cpp
template <typename Ret, typename... Param>
void add(std::string name, Ret (*func)(Param...)) {
  functions.try_emplace(
      std::move(name),
      [func](std::span<std::any> params) {
        return [&]<std::size_t... Index>(std::index_sequence<Index...>)->std::any {
          if constexpr (!std::is_same_v<void, Ret>) /// not void
            return func(std::any_cast<Param>(params[Index])...);
          } else { /// void (would [[likely]] help here?
            func(std::any_cast<Param>(params[Index])...);
            return {};
          }
        }(std::make_index_sequence<sizeof...(Param)>());
      },
      sizeof...(Param));
}

void print(int);
void print(std::string);

int main() {
  add("print_string", static_cast<void (*)(std::string)>(print));
  add("print_int", static_cast<void (*)(int)>(print));
  add<void, int>("print_alt", print); /// OR
}
```

https://godbolt.org/z/5aGPdj6Y8

Executing Functions

Executing Functions

```
1  void exec(const std::string &func_name) {
2    const auto func = functions.at(func_name);
3    const auto begin = std::prev(stack.end(),
4                                 static_cast<std::ptrdiff_t>(func.arity));
5
6    auto result = func.callable(std::span{begin, stack.end()}); /// 1 Get result
7
8    stack.erase(begin, stack.end()); /// 2 erase used values
9
10   if (result.has_value()) {
11     /// I don't like this, but don't have a better option
12     /// we have to query the value to know if we want to keep it
13     stack.push_back(std::move(result));
14   }
15 }
```

https://godbolt.org/z/8W9vsrfoe

Parsing The Script Input

Parsing The Script Input

```
1  void parse(std::string input) {
2    // Empty
3    if (input.empty()) { return; }
4
5    // Function to execute
6    if (functions.contains(input)) {
7      exec(input);
8      return;
9    }
10
11   // Possible number
12   const auto begin = input.data();
13   const auto end = std::next(input.data(), std::ssize(input));
14   int value = 0;
15   if (auto result = std::from_chars(begin, end, value); result.ptr == end) {
16     stack.emplace_back(value);
17     return;
18   }
19
20   // Otherwise string
21   stack.emplace_back(std::move(input));
22 }
```

https://godbolt.org/z/7z6evaac1

Putting It Together

Putting It Together

Link

https://compiler-explorer.com/z/dzofzoros

Bonus - Overloading

Overloading

What are our options, if any?

Overloading

What are our options, if any?

- Arity-based overloading
  - This is usually the “easier” option, but for our stack-based interpreter it’s harder
  - How do we decide how many parameters to take?

Overloading

What are our options, if any?

- Arity-based overloading
  - This is usually the “easier” option, but for our stack-based interpreter it’s harder
  - How do we decide how many parameters to take?
- Type-based overloading (effectively 2 options)
  - Just try every function with a given name and see which succeeds
  - Store a type-list for each overload and do a run-time check to filter possible overloads

Overloading

This is hard, and potentially expensive. Tools like SWIG create a single proxy function for the target language that knows how to dispatch to the overloads.

Overloading

Knowing what you know now, you should be able to interpret this snippet from some SWIG generated code…

```
1  if (argc == 1) {
2      int _v;
3      {
4          int res = SWIG_AsVal_int(argv[0], NULL);
5          _v = SWIG_CheckState(res);
6      }
7      if (_v) {
8          return _wrap__print__SWIG_0(self, args);
9      }
10 }
11
12 if (argc == 1) {
13     int _v;
14     int res = SWIG_AsPtr_std_string(argv[0], (std::string**)0);
15     _v = SWIG_CheckState(res);
16     if (_v) {
17         return _wrap__print__SWIG_1(self, args);
18     }
19 }
```

https://godbolt.org/z/z4Gzv3M38

Who wants to give it a go?

Your Homework

Your Homework

Try to add overloading support.

Your Homework

Add more structure to the scripting language.

Your Homework

Add member function support and member object support.

Your Homework

See what reflection would gain you, if anything.

Your Homework

Remember to test, particularly with fuzz testing. It's easy to create dangerous scenarios when parsing user input.

```
1  int main() {
2    add("process", &function_expecting_20_parameters);
3    // add 21 parameters to stack
4    while (!stack.empty()) {
5      call("process");
6    }
7  }
```

https://godbolt.org/z/5hqc8KEaW

Your Homework

CE version of this script engine that has had some enhancements and TODO items.

https://compiler-explorer.com/z/86558dW56

Your Homework

gist of this script engine that has had some enhancements and TODO items.

https://gist.github.com/lefticus/5d94357725413dce5005b0b1b7f77836

Secrets of Scripting Bindings for C++

Jason Turner

C++ Weekly

- Weekly videos since March, 2016
- 112k+ subscribers, 445+ weeks straight

https://www.youtube.com/@cppweekly

Jason Turner

- Author
  - C++ Best Practices, C++23 Best Practices
  - OpCode, Copy and Reference, Object Lifetime Puzzlers
  - https://amzn.to/3xWh8Ox
  - https://leanpub.com/u/jason_turner

Jason Turner

- Developer
  - https://cppbestpractices.com
  - https://github.com/lefticus
  - https://github.com/cpp-best-practices
- Microsoft MVP for C++, 2015-present

Jason Turner - Training

https://articles.emptycrate.com/training.html

How to get my training:

1. Have me come to your company on-site for dynamic customized training where you already are - generally the most economical option for groups (DE, NL, RO, CZ, JP, US, PL, SE, …)
2. Come to a conference workshop
   - C++ On Sea (Folkestone, UK, ~July)
   - CppCon (Aurora, CO, US, ~Sept)
   - NDC TechTown (Kongsberg, NO, ~Sept)
   - C++ Under the Sea (Breda, NL, ~Oct)
   - And possibly others