Surprises In Object Lifetime

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

About Surprises In Object Lifetime

- I’ve taught a 1 day course called “Understanding Object Lifetime” ~12 times
- Well-defined object lifetime, this construction/destruction cycle is a key feature of C++
- Few other languages let us reason about object lifetime in the same way
- But sometimes there are surprises, sometimes for the better, sometimes not

What is an Object?

How is an `int` different from something containing an `int`?

```cpp
// how is
struct S { int i; };
// different from
int i;
// ?
```

With uniform initializaton syntax the differences are not so clear

```cpp
struct S { int i; };

int use_s() {
  static_assert(sizeof(S) == sizeof(int));
  S s{15};
  int &i = reinterpret_cast<int&>(s); // don't do this in real code
  i=23;
  return s.i;
}
```

```cpp
int use_int() {
  static_assert(sizeof(int) == sizeof(int));
  int s{15};
  int &i = reinterpret_cast<int&>(s);
  i=23;
  return i;
}
```

Both are

```cpp
static_assert(std::is_trivially_constructible_v<S>);
static_assert(std::is_trivially_constructible_v<int>);
static_assert(std::is_trivially_destructible_v<S>);
static_assert(std::is_trivially_destructible_v<int>);
static_assert(std::is_trivially_copyable_v<S>);
static_assert(std::is_trivially_copyable_v<int>);
static_assert(std::is_object_v<S>);
static_assert(std::is_object_v<int>);
```

What does the standard say?

[basic.types (8)]

> *An object type is a (possibly cv-qualified) type that is not a function type, not a reference type, and not cv void.*

Object Lifetime

Lifetime Begins

[basic.life]

> *The lifetime of an object of type T begins when:*
>
> *(1.1) storage with the proper alignment and size for type T is obtained, and*
>
> *(1.2) if the object has non-vacuous initialization, its initialization is complete, except that if the object is a union member or subobject thereof, its lifetime only begins if that union member is the initialized member in the union (11.6.1, 15.6.2), or as described in 12.3*

Lifetime Ends

[basic.life]

> *The lifetime of an object o of type T ends when:*
>
> *(1.3) if T is a class type with a non-trivial destructor (15.4), the destructor call starts, or*
>
> *(1.4) the storage which the object occupies is released, or is reused by an object that is not nested within o (4.5).*

Basic Object Lifetime

What’s Printed?

```cpp
#include <cstdio>
struct S {
  S() { puts("S()"); }
  S(const S &) noexcept { puts("S(const S &)"); }
  S(S &&) noexcept { puts("S(S&&)"); }
  S &operator=(const S &) { puts("operator=(const S&)"); return *this; }
  S &operator=(S &&) { puts("operator=(S&&)"); return *this; }
  ~S() { puts("~S()"); }
};

int main() {
  S s;

  {
    S s2{s};
  }
}
```

Basic Object Lifetime

What’s Printed?

```cpp
#include <cstdio>
struct S {
  S() { puts("S()"); }
  S(const S &) noexcept { puts("S(const S &)"); }
  S(S &&) noexcept { puts("S(S&&)"); }
  S &operator=(const S &) { puts("operator=(const S&)"); return *this; }
  S &operator=(S &&) { puts("operator=(S&&)"); return *this; }
  ~S() { puts("~S()"); }
};

int main() {
  S s;  /// S()

  {
    S s2{s}; /// S(const S &)
  }          /// ~S()
}       /// ~S()
```

Basic Object Lifetime

What’s Printed?

```
#include <cstdio>
struct S {
  S() { puts("S()"); }
  S(const S &) noexcept { puts("S(const S &)"); }
  S(S &&) noexcept { puts("S(S&&)"); }
  S &operator=(const S &) { puts("operator=(const S&)"); return *this; }
  S &operator=(S &&) { puts("operator=(S&&)"); return *this; }
  ~S() { puts("~S()"); }
};

int main() {
  S s;

  {
    [[maybe_unused]] S &s2{s};
  }
}
```

Basic Object Lifetime

What's Printed?

```cpp
#include <cstdio>
struct S {
  S() { puts("S()"); }
  S(const S &) noexcept { puts("S(const S &)"); }
  S(S &&) noexcept { puts("S(S&&)"); }
  S &operator=(const S &) { puts("operator=(const S&)"); return *this; }
  S &operator=(S &&) { puts("operator=(S&&)"); return *this; }
  ~S() { puts("~S()"); }
};

int main() {
  S s;  /// S()

  {
    [[maybe_unused]] S &s2{s};
  }
}         /// ~S()
```

Remember that `&` types are not object types, they have no lifetime.

What is printed?

```cpp
#include <iostream>

const int & get_data() {
  const int i = 5;
  return i;
}

int main() {
  std::cout << get_data();
}
```

What is printed?

Unknown!

Ok, what warning do we get?

`reference to stack memory returned` (or similar)

Are we surprised yet? (Probably not)

What is printed?

```cpp
#include <iostream>
#include <functional>

std::reference_wrapper<const int> get_data() {
  const int i = 5;
  return i;
}

int main() {
  std::cout << get_data();
}
```

What is printed?

Unknown!

Ok, what warning do we get?

What Warning Do We Get?

(clang and gcc differ here)

```cpp
#include <iostream>
#include <functional>

std::reference_wrapper<const int> get_data() {
  const int i = 5;
  return i;
}

int main() {
  std::cout << get_data();
}
```

Surprise!

Simple standard library wrappers around references confuse analysis.

Strings

What’s printed?

```
#include <string>
#include <iostream>

const char * get_data()
{
  return "Hello World";
}

int main()
{
  std::cout << get_data();
}
```

What’s printed?

```
Hello World
```

Why is this allowed?

String Literals

[lex.string]

Evaluating a string-literal results in a string literal object with static storage duration, initialized from the given characters as specified above.

(`static` objects are valid for the entirety of the program)

What’s Printed?

```
#include <string>
#include <iostream>

std::string_view get_data()
{
  return "Hello World";
}

int main()
{
  std::cout << get_data();
}
```

What's Printed?

`Hello World`

This is not surprising, `std::string_view` is a pair of pointers into the statically constructed `"Hello World"` string.

What's Printed?

```cpp
#include <string>
#include <iostream>

std::string_view get_data()
{
  std::string s = "Hello World";
  return s;
}

int main()
{
  std::cout << get_data();
}
```

What’s Printed?

Unknown

Why?

`std::string_view` return value now points into the data which was owned by the local `std::string`.

What warnings do we get?

What Warnings Do We Get?

```cpp
#include <string>
#include <iostream>

std::string_view get_data()
{
  std::string s = "Hello World";
  return s;
}

int main()
{
  std::cout << get_data();
}
```

One Last Thing To Say About Strings

What happens here?

```cpp
#include <string>
#include <iostream>

std::string_view get_data()
{
  const char s[] = "Hello World";
  return s;
}

int main()
{
  std::cout << get_data();
}
```

One Last Thing To Say About Strings

What happens here?

```cpp
#include <string>
#include <iostream>

std::string_view get_data()
{
  const char s[] = "Hello World"; /// local array
  return s;
}

int main()
{
  std::cout << get_data();
}
```

One Last Thing To Say About Strings

What happens here?

```cpp
#include <string>
#include <iostream>

std::string_view get_data()
{
  const char s[] = "Hello World";
  return s; /// decays to pointer, inits string_view
}

int main()
{
  std::cout << get_data();
}
```

One Last Thing To Say About Strings

What happens here?

```cpp
#include <string>
#include <iostream>

std::string_view get_data()
{
  const char s[] = "Hello World";
  return s;
}

int main()
{
  std::cout << get_data(); /// no warnings
}
```

Surprise!

Strings live longer than you think they will, except when they don’t.

Containers

What is Printed?

```
#include <cstdio>
#include <vector>

struct S {
  S() { puts("S()"); }
  S(int) { puts("S(int)"); }
  S(const S &) noexcept { puts("S(const S &)"); }
  S(S &&) noexcept { puts("S(S&&)"); }
  S &operator=(const S &) { puts("operator=(const S&)"); return *this; }
  S &operator=(S &&) { puts("operator=(S&&)"); return *this; }
  ~S() { puts("~S()"); }
};

int main() {
  std::vector<S> vec;
  vec.push_back(S{1});
}
```

What is Printed?

```
S(int)
S(S&&)
~S()
~S()
```

Remember that for a non-trivially destructible type, the destructor of a moved-from object must still be called.

Surprise!

Moved-from objects still have to be destroyed. For non-trivial types, this is non-trivial, often inlining a destructor (or maybe worse, not inlining it) that has to still check a pointer to see if resources need to be cleaned up.

What is Printed?

```cpp
#include <cstdio>
#include <vector>

struct S {
  S() { puts("S()"); }
  S(int) { puts("S(int)"); }
  S(const S &) noexcept { puts("S(const S &)"); }
  S(S &&) noexcept { puts("S(S&&)"); }
  S &operator=(const S &) { puts("operator=(const S&)"); return *this; }
  S &operator=(S &&) { puts("operator=(S&&)"); return *this; }
  ~S() { puts("~S()"); }
};

int main() {
  std::vector<S> vec;
  vec.emplace_back(S{int}); ///
}
```

What is Printed?

```
S(int)
S(S&&)
~S()
~S()
```

Same thing as `push_back`. This is an improper use of `emplace_back`. The point of `emplace_back` is that you are directly calling the constructor of the object you want to add to the container.

What is Printed?

```cpp
#include <cstdio>
#include <vector>

struct S {
  S() { puts("S()"); }
  S(int) { puts("S(int)"); }
  S(const S &) noexcept { puts("S(const S &)"); }
  S(S &&) noexcept { puts("S(S&&)"); }
  S &operator=(const S &) { puts("operator=(const S&)"); return *this; }
  S &operator=(S &&) { puts("operator=(S&&)"); return *this; }
  ~S() { puts("~S()"); }
};

int main() {
  std::vector<S> vec;
  vec.emplace_back(1); ///
}
```

What is Printed?

```
S(int)
~S()
```

This is the correct way to use `emplace_back` and specifically in this case, with a default constructed object.

What is Printed?

```cpp
#include <cstdio>
#include <vector>

struct S {
  S() { puts("S()"); }
  S(int) { puts("S(int)"); }
  S(const S &) noexcept { puts("S(const S &)"); }
  S(S &&) noexcept { puts("S(S&&)"); }
  S &operator=(const S &) { puts("operator=(const S&)"); return *this; }
  S &operator=(S &&) { puts("operator=(S&&)"); return *this; }
  ~S() { puts("~S()"); }
};

int main() {
  std::vector<S> vec;
  vec.emplace_back(); /// note that this works for 0 params too
}
```

Surprise!

Even without a named object we have to be thinking about lifetime!

Temporaries

What's Printed?

```cpp
#include <cstdio>

struct S {
  S() { puts("S()"); }
  ~S() { puts("~S()"); }
};

S get_value() { return {}; }

int main()
{
  const auto &val = get_value();
  puts("Hello World");
}
```

What's Returned?

```
S()
Hello World
~S()
```

Surprise!

Complex rules allow for the lifetime extension of temporaries that are assigned to references (See: [class.temporary])

What's Returned From Main?

```cpp
struct S {
  const int& m;
};

int main() {
  const S& s = S{1};
  return s.m;
}
```

What’s Returned From Main?

```
1
```

Surprise!

Lifetime extension rules apply recursively to member initializers

Initializer Lists

`std::initializer_list<>`

```cpp
// how many dynamic allocations are there?
std::vector<std::string> vec{"a", "b"};
```

Almost certainly only 1. Every standard library implements a “Small String Optimization,” so only the `vector` needs an allocation.

Surprise!

> `std::string` *is highly optimized, don't underestimate it.*

std::initializer_list<>

```cpp
// how many dynamic allocations are there?
std::vector<std::string> vec{"a long string of characters",
                             "b long string of characters"};
```

5

Why?

`std::initializer_list<>`

`initializer_list` is implemented by creating a hidden array for you, of the expected type, that is `const`.

So this is the approximate equivalence.

```
const std::string __data[]{"a long string of characters",
                           "b long string of characters"};
std::vector<std::string> vec{initializer_list<std::string>{__data,
                                                           __data + 2}};
```

std::initializer_list<>

```
const std::string __data[]{"a long string of characters",  // alloc 1
                           "b long string of characters"}; // alloc 2
// vector: alloc 3
// copy of str1: alloc 4
// copy of str2: alloc 5
std::vector<std::string> vec{initializer_list<std::string>{__data,
                                                           __data + 2}};
```

Surprise!

`std::initializer_list<>` *invocations create hidden* `const` *arrays.*

`std::array<>`

```cpp
// how many dynamic allocations are done?
// (C++17 class template type deduction)
std::array a{"a long string of characters", "b long string of characters"};
```

0 … Why?

Type of `std::array` is deduced as `std::array<const char *, 2>`.

`std::array<>`

```
// Is this OK?
std::array a{"a long string of characters", "b long string of characters"};
```

Sure, we know the character string literal will be valid for the life of the program.

`std::array<>`

```cpp
// how many dynamic allocations are done?
std::array<std::string, 2> a{"a long string of characters",
                             "b long string of characters"};
```

Exactly 2.

What’s the difference?

`std::array<>`

`std::array<>` has no constructors, it is effectively something like:

```
template<typename T, std::size_t Size>
struct array
{
  T _M_elems[Size];
};
```

So with `std::array` our “Initializer List” initialization does not use an `initializer_list<>`, it *directly initializes* the internal data structure.

This is literally the most efficient thing possible!

Surprise!

`std::array` *has 0 constructors, for efficiency!*

Ranged `for` Loops

What’s Printed?

```cpp
#include <vector>
#include <iostream>

struct S {
  std::vector<int> data{1,2,3,4,5};
  const auto &get_data() const { return data; }
};

S get_s() {  return S{}; }

int main() {
  for (const auto &v : get_s().get_data()) {
    std::cout << v;
  }
}
```

What’s Printed?

Unknown!

Why?

What’s Printed? - Equiv

```cpp
#include <vector>
#include <iostream>

struct S {
  std::vector<int> data{1,2,3,4,5};
  const auto &get_data() const { return data; }
};

S get_s() {  return S{}; }

int main() {
  {
    auto &&__range = get_s().get_data();
    auto __begin = begin(__range);
    auto __end = end(__range);
    for ( ; __begin != __end; ++__begin ) {
      const auto &v = *__begin;
      std::cout << v;
    }
  }
}
```

What’s Printed? - Equiv

```
#include <vector>
#include <iostream>

struct S {
  std::vector<int> data{1,2,3,4,5};
  const auto &get_data() const { return data; }
};

S get_s() {  return S{}; }

int main() {
  {
    auto &&__range = get_s().get_data(); /// dangling reference
    auto __begin = begin(__range);
    auto __end = end(__range);
    for ( ; __begin != __end; ++__begin ) {
      const auto &v = *__begin;
      std::cout << v;
    }
  }
}
```

What Warnings Do We Get?

```cpp
#include <vector>
#include <iostream>

struct S {
  std::vector<int> data{1,2,3,4,5};
  const auto &get_data() const { return data; }
};

S get_s() {  return S{}; }

int main() {
  for (const auto &v : get_s().get_data()) {
    std::cout << v;
  }
}
```

C++20 for-init

```cpp
#include <vector>
#include <iostream>

struct S {
  std::vector<int> data{1,2,3,4,5};
  const auto &get_data() const { return data; }
};

S get_s() {  return S{}; }

int main() {
  for (const auto s = get_s(); ///
       const auto &v : s.get_data()) {
    std::cout << v;
  }
}
```

C++20 for-init: equiv

```cpp
#include <vector>
#include <iostream>

struct S {
  std::vector<int> data{1,2,3,4,5};
  const auto &get_data() const { return data; }
};

S get_s() {  return S{}; }

int main() {
  {
    const auto s = get_s(); /// init statement
    auto &&__range = s.get_data(); /// no dangling reference
    auto __begin = begin(__range);
    auto __end = end(__range);
    for ( ; __begin != __end; ++__begin ) {
      const auto &v = *__begin;
      std::cout << v;
    }
  }
}
```

Surprise!

Ranged-`for` loops create hidden variables for you that have their own lifetime questions.

On The Topic of -init Statements, if-init

if-init

What warning might this give?

```
int get_val();
double get_other_val();

int main() {
  if (const auto x = get_val(); x > 5) {
    // do something with x
  } else if (const auto x = get_other_val(); x < 5) {
    // do something else with x
  }
}
```

`x shadows previous declaration of x` (or similar)

if-init - equiv

```cpp
int get_val();
double get_other_val();

int main() {
  if (const auto x = get_val(); x > 5) {
    // do something with x
  } else if (const auto x = get_other_val(); x < 5) {
    // do something else with x
  }
}
```

if-init - equiv

```
int get_val();
double get_other_val();

int main() {
  {
    const auto x = get_val();
    if (x > 5) {
      // do something with x
    } else if (const auto x = get_other_val(); x < 5) {
      // do something else with x
    }
  }
}
```

if-init - equiv

```cpp
int get_val();
double get_other_val();

int main() {
  {
    const auto x = get_val();
    if (x > 5) {
      // do something with x
    } else {
      const auto x = get_other_val(); /// shadowing
      if (x < 5) {
        // do something else with x
      }
    }
  }
}
```

Surprise!

If-init statements are visible for the `else` *blocks as well*

RVO

What’s Printed?

```cpp
#include <cstdio>

struct S {
  S() { puts("S()"); }
  S(const S &) noexcept { puts("S(const S &)"); }
  S(S &&) noexcept { puts("S(S&&)"); }
  S &operator=(const S &) { puts("operator=(const S&)"); return *this; }
  S &operator=(S &&) { puts("operator=(S&&)"); return *this; }
  ~S() { puts("~S()"); }
};

S get_S() {
  return {};
}

int main() {
  get_S();
}
```

What’s Printed?

```
S()
~S()
```

What’s Printed?

```cpp
#include <cstdio>

struct S {
  S() { puts("S()"); }
  S(const S &) noexcept { puts("S(const S &)"); }
  S(S &&) noexcept { puts("S(S&&)"); }
  S &operator=(const S &) { puts("operator=(const S&)"); return *this; }
  S &operator=(S &&) { puts("operator=(S&&)"); return *this; }
  ~S() { puts("~S()"); }
};

S get_S() { return {}; }
S get_other_S() { return get_S(); }

int main() {
  S s = get_other_S(); ///
}
```

What’s Printed?

```
S()
~S()
```

Required as of C++17, but true in every compiler going back to at least 1995.

Surprise!

RVO is super awesome, rely on it.

Subobjects

What’s Printed?

```cpp
#include <cstdio>

struct S {
  S() { puts("S()"); }
  S(const S &) noexcept { puts("S(const S &)"); }
  S(S &&) noexcept { puts("S(S&&)"); }
  S &operator=(const S &) { puts("operator=(const S&)"); return *this; }
  S &operator=(S &&) { puts("operator=(S&&)"); return *this; }
  ~S() { puts("~S()"); }
};

struct Holder { S s; int i };

Holder get_Holder() { return {}; }
S get_S() {
  S s = get_Holder().s;
  return s;
}

int main() {
  S s = get_S();
}
```

What’s Printed?

```
S()
S(S&&)
~S()
~S()
```


What’s Printed?

```cpp
#include <cstdio>

struct S {
  S() { puts("S()"); }
  S(const S &) noexcept { puts("S(const S &)"); }
  S(S &&) noexcept { puts("S(S&&)"); }
  S &operator=(const S &) { puts("operator=(const S&)"); return *this; }
  S &operator=(S &&) { puts("operator=(S&&)"); return *this; }
  ~S() { puts("~S()"); }
};

struct Holder { S s; int i };

Holder get_Holder() { return {}; } /// init Holder, S()
S get_S() {
  S s = get_Holder().s; /// r-value inits s, S(S&&), ~S() of moved from
  return s; /// rvo applied
}

int main() {
  S s = get_S();  /// nothing printed
}  /// ~S() from this `s`
```

Surprise!

Moves happen automatically with r-values. No need to help the compiler.

Structured Bindings

What’s Printed?

```cpp
#include <cstdio>

struct S {
  S() { puts("S()"); }
  S(const S &) noexcept { puts("S(const S &)"); }
  S(S &&) noexcept { puts("S(S&&)"); }
  S &operator=(const S &) { puts("operator=(const S&)"); return *this; }
  S &operator=(S &&) { puts("operator=(S&&)"); return *this; }
  ~S() { puts("~S()"); }
};

struct Holder { S s; int i };

Holder get_Holder() { return {}; }
S get_S() {
  auto [s, i] = get_Holder(); /// structured bindings
  return s;
}

int main() {
  S s = get_S();
}
```

What’s Printed?

```
S()
S(const S&) /// copy, not move now
~S()
~S()
```

What’s Printed? - Equiv

```cpp
#include <cstdio>

struct S {
  S() { puts("S()"); }
  S(const S &) noexcept { puts("S(const S &)"); }
  S(S &&) noexcept { puts("S(S&&)"); }
  S &operator=(const S &) { puts("operator=(const S&)"); return *this; }
  S &operator=(S &&) { puts("operator=(S&&)"); return *this; }
  ~S() { puts("~S()"); }
};

struct Holder { S s; int i };

Holder get_Holder() { return {}; }
S get_S() {
  auto e = get_Holder(); ///
  auto &s = e.s;         ///
  auto &i = e.i;         ///
  return s;              /// RVO not applied to reference
}

int main() {
  S s = get_S();
}
```

Surprise!

Structured bindings create hidden values that are references (which aren’t objects) so RVO and automatic moves and such cannot happen.

Delegating Constructors

Is The Destructor Called?

```cpp
#include <cstdio>

struct S {
  int i{};
  S() = default;
  S(int i_) : i{i_} {}
  ~S() { puts("~S()"); }
};

int main() {
  try {
    S s{1};
  } catch (...) {
  }
}
```

Yes.

Is The Destructor Called?

```cpp
#include <cstdio>

struct S {
  int i{};
  S() = default;
  S(int i_) : i{i_}
  { throw 1; } ///
  ~S() { puts("~S()"); }
};

int main() {
  try {
    S s{1};
  } catch (...) {
  }
}
```

No.

Is The Destructor Called?

```cpp
#include <cstdio>

struct S {
  int i{};
  S() = default;
  S(int i_) : i{i_}
  { throw 1; } /// constructor doesn't complete
  ~S() { puts("~S()"); }
};

int main() {
  S s{1};
}
```

Is The Destructor Called?

```cpp
#include <cstdio>

struct S {
  int i{};
  S() = default;
  S(int i_)
    : S{} ///
  { i = i_; throw 1; }
  ~S() { puts("~S()"); }
};

int main() {
  try {
    S s{1};
  } catch (...) {
  }
}
```

Yes!

Why?

Is The Destructor Called?

```
#include <cstdio>

struct S {
  int i{};
  S() = default;
  S(int i_)
    : S{} /// Once delegating constructor completes
          /// the object's lifetime has begun
  { i = i_; throw 1; }
  ~S() { puts("~S()"); }
};

int main() {
  try {
    S s{1};
  } catch (...) {
  }
}
```

Is The Destructor Called?

Can be used in interesting ways (From Howard Hinnant)

```cpp
struct S {
  int *ptr{nullptr};
  int *ptr2{nullptr};
  S() = default;
  S(int val1, int val2) : S{} /// make sure d'tor is called
  {
    ptr = new int(val1);
    ptr2 = new int(val2);
  }
  ~S() { delete ptr; delete ptr2; } /// delete nullptr is well defined
};
```

OF COURSE DON’T DO THIS, USE unique_ptr instead

Surprise!

An object's lifetime has begun after any constructor has completed.

Anywhere where the specs say the compiler transformed the code, there might be a surprise lurking.

Avoiding Lifetime Issues

Think About Lifetime

Don’t Name Temporaries

```cpp
auto get_first()
{
  auto [first, second] = get_pair();
  return first; // bad idea
}
```

```cpp
auto get_first()
{
  return get_pair().first; /// good idea
}
```

Consider Requiring All Structured Bindings To Be `&`

```cpp
auto get_sum()
{
  // const & works well with lifetime extension rules
  // and makes it clear we are actually playing with
  // hidden references
  const auto &[first, second] = get_pair();
  return first + second;
}
```

Use The Tools

Warn All The Things

- `-Wshadow`
- MSVC and clang-tidy core guideline checks for things like array to pointer conversion

Sanitize!

All of these examples which could cause a crash are trivial to catch with tests that use sanitizers.

Carefully Use `initializer_list<>`

- Understand the difference between an *Initializer List* and an `initializer_list<>` (note that [dcl.init.list] has 12 subclauses)
- Take advantage of direct initialization for type safety and performance
- Only use `initializer_list<>` constructors for `trivial` or `literal` types

`constexpr` All The Things

What would this return?

```
int & get_val() {
  int i{};
  return i; /// dandling reference
}

int do_thing() {
  return ++get_val(); /// invalid dereference
}

int main() {
  auto val = do_thing();
  return val; /// unknown
}
```

`constexpr` All The Things

`constexpr` doesn’t allow undefined behavior. Compiler enforcement varies.

```
constexpr int & get_val() {
  int i{};
  return i;
}

constexpr int do_thing() {
  return ++get_val();
}

int main() {
  constexpr auto val = do_thing();
  return val;
}
```

What’s Printed?

```cpp
#include <array>
#include <iostream>
struct S {
  const std::array<int,5> data{1,2,3,4,5};
  const auto &get_data() const { return data; }
};
S get_s() {  return S{}; }


int sum() {
  int i = 0;
  for (const auto &v : get_s().get_data()) { /// dangling ref
    i += v;
  }
  return i;
}

int main() {
  const int s = sum(); /// unknown
  std::cout << s;
}
```

What’s Printed?

`constexpr` version…

```
#include <array>
#include <iostream>
struct S {
  const std::array<int,5> data{1,2,3,4,5};
  constexpr const auto &get_data() const { return data; }
};
constexpr S get_s() {  return S{}; }

constexpr int sum() {
  int i = 0;
  for (const auto &v : get_s().get_data()) {
    i += v;
  }
  return i;
}

int main() {
  constexpr const int s = sum();
  std::cout << s;
}
```

What’s Printed?

```cpp
#include <string>
#include <iostream>

std::string_view get_value()
{
  const char str[] = "Hello World";
  return str;
}

int main()
{
  auto sv = get_value();
  std::cout << sv;
}
```

What’s Printed?

```cpp
#include <string>
#include <iostream>

constexpr std::string_view get_value()
{
  const char str[] = "Hello World";
  return str;
}

int main()
{
  constexpr auto sv = get_value(); /// won't compile now
  std::cout << sv;
}
```

Jason Turner

- First used C++ in 1996

Copyright Jason Turner @lefticus

- 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

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