Great C++ `is_trivial`

Jason Turner

- Co-host of CppCast https://cppcast.com
- Host of C++ Weekly https://www.youtube.com/c/JasonTurner-lefticus
- Projects
  - https://chaiscript.com
  - https://cppbestpractices.com
  - https://github.com/lefticus/cpp_box
  - https://coloradoplusplus.info
- Microsoft MVP for C++ 2015-present

Jason Turner

Independent and available for training or contracting

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

Check out the “North Denver Metro C++ Meetup,” we’ve been meeting consistently since November 2016!

About my Talks

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

Upcoming Events

- CppCon - Sept 21, 2019 - Applied `constexpr` - Doing More At Compile-Time

What Do We Mean By Great?

What Do We Mean By Great?

- Easier to write?

What Do We Mean By Great?

- Easier to write?
- Easier to maintain?

What Do We Mean By Great?

- Easier to write?
- Easier to maintain?
- More optimizable?

What Do We Mean By Great?

- Easier to write?
- Easier to maintain?
- More optimizable?

We'll touch a bit on each of these things, but focus on the parts that let the optimizer work.

What Do We Mean By Great?

- Easier to write?
- Easier to maintain?
- More optimizable?

We'll touch a bit on each of these things, but focus on the parts that let the optimizer work.

This is not a "Best Practices" talk per se, it's a talk to make you think more about your `class`es.

What Do We Mean By Great?

- Easier to write?
- Easier to maintain?
- More optimizable?

We'll touch a bit on each of these things, but focus on the parts that let the optimizer work.

This is not a "Best Practices" talk per se, it's a talk to make you think more about your `class`es.

We will build a little bit on the things discussed on Tuesday in "C++ Code Smells."

Efficiency With Values

Return string by value / move

```cpp
std::string get_val() { // A
  std::string val{"Hello There World!"};
  return std::move(val);
}
```
https://godbolt.org/z/PopltJ

```cpp
std::string get_val() { // B
  std::string val{"Hello There World!"};
  return val;
}
```
https://godbolt.org/z/RIUgqM

```cpp
std::string get_val() { // C
  const std::string val{"Hello There World!"};
  return std::move(val);
}
```
https://godbolt.org/z/M0Rer-

```cpp
std::string get_val() { // D
  const std::string val{"Hello There World!"};
  return val;
}
```
https://godbolt.org/z/Nddmew

Return string by value / move

```cpp
std::string get_val() { // A
  std::string val{"Hello There World!"};
  return std::move(val); //  Move forced, bad
}
```

https://godbolt.org/z/S8nj_7

```cpp
std::string get_val() { // B
  std::string val{"Hello There World!"};
  return val; /// copy/move elision, good
}
```

https://godbolt.org/z/kMGhX-

```cpp
std::string get_val() { // C
  const std::string val{"Hello There World!"};
  return std::move(val); //  copy forced, bad
}
```

https://godbolt.org/z/djHntA

```cpp
std::string get_val() { // D
  const std::string val{"Hello There World!"};
  return val; /// copy/move elision, good
}
```

https://godbolt.org/z/-nPv_f

Return string by value / move

```
1  std::string get_val() { // A
2    auto [str1, str2] = get_string_pair();
3    return std::move(str1);
4  }
```
https://godbolt.org/z/GAIZDY

```
1  std::string get_val() { // B
2    auto [str1, str2] = get_string_pair();
3    return str1;
4  }
```
https://godbolt.org/z/PUrQFJ

```
1  std::string get_val() { // C
2    auto &&[str1, str2] = get_string_pair();
3    return str1;
4  }
```
https://godbolt.org/z/w-3TYK

```
1  std::string get_val() { // D
2    const auto [str1, str2] = get_string_pair();
3    return std::move(str1);
4  }
```
https://godbolt.org/z/piQrYi

```
1  std::string get_val() { // E
2    const auto [str1, str2] = get_string_pair();
3    return str1;
4  }
```
https://godbolt.org/z/KE_MKS

Return string by value / move

```
1  std::string get_val() { // A
2    auto [str1, str2] = get_string_pair();
3    return std::move(str1); /// allows move, OK
4  }
```
https://godbolt.org/z/7nC5-k

```
1  std::string get_val() { // B
2    auto [str1, str2] = get_string_pair();
3    return str1; //  Copy, not OK
4  }
```
https://godbolt.org/z/OyWBXU

```
1  std::string get_val() { // C
2    auto &&[str1, str2] = get_string_pair();
3    return str1; /// Implicit move?? P18250
4  }
```
https://godbolt.org/z/VGhMmK

```
1  std::string get_val() { // D
2    const auto [str1, str2] = get_string_pair();
3    return std::move(str1); //  copy
4  }
```
https://godbolt.org/z/6qGZfl

```
1  std::string get_val() { // E
2    const auto [str1, str2] = get_string_pair();
3    return str1; //  copy
4  }
```
https://godbolt.org/z/RQ001c

Pass By Value / Move

```
1  void do_things() { // A
2    std::string str{"Hello There World"};
3    use_string(str);
4  }
```

https://godbolt.org/z/Tk-jLk

```
1  void do_things() { // B
2    std::string str{"Hello There World"};
3    use_string(std::move(str));
4  }
```

https://godbolt.org/z/wDVyx_

```
1  void do_things() { // C
2    use_string("Hello There World");
3  }
```

```
1  void do_things() { // D
2    use_string(get_string());
3  }
```

Pass By Value / Move

```cpp
void do_things() { // A
  std::string str{"Hello There World"};
  use_string(str); //  possible copy
}
```

https://godbolt.org/z/KWwa0_

```cpp
void do_things() { // B
  std::string str{"Hello There World"};
  use_string(std::move(str)); //  possible move
}
```

https://godbolt.org/z/p8h-a3

```cpp
void do_things() { // C
  use_string("Hello There World"); /// direct-init of param
}
```

```cpp
void do_things() { // D
  use_string(get_string()); /// direct-init of param
}
```

Out Parameter vs Return Value

```
1  void use_value() { // A
2    std::string s = get_value();
3  }
```

```
1  void use_value() { // B
2    const std::string s = get_value();
3  }
```

```
1  void use_value() { // C
2    std::string s;
3    get_value(s);
4  }
```

https://godbolt.org/z/qYDxRi

```
1  void use_value() { // D
2    const std::string s;
3    get_value(s);
4  }
```

https://godbolt.org/z/VZNCGN

Out Parameter vs Return Value

```
1  void use_value() { // A
2    std::string s = get_value(); /// direct-init/RVO
3  }
```

```
1  void use_value() { // B
2    const std::string s = get_value(); /// direct-init/RVO
3  }
```

```
1  void use_value() { // C
2    std::string s; //  default construct
3    get_value(s);  //  assign
4  }
```

https://godbolt.org/z/LxQZ4x

```
1  void use_value() { // D
2    const std::string s; //  default construct
3    get_value(s);        //  cannot compile
4  }
```

https://godbolt.org/z/ruiGqf

Assignment Vs Initialization

```
1  std::string get_value(); // A
2
3  void use_value() {
4    std::string s;
5    s = get_value();
6  }
```

https://godbolt.org/z/k0jipF

```
1  const std::string get_value(); // B
2
3  void use_value() {
4    std::string s;
5    s = get_value();
6  }
```

https://godbolt.org/z/VftllZ

```
1  std::string get_value(); // C
2
3  void use_value() {
4    const std::string s = get_value();
5  }
```

https://godbolt.org/z/htw1YN

```
1  const std::string get_value(); // D
2
3  void use_value() {
4    std::string s = get_value();
5  }
```

https://godbolt.org/z/VWW17V

Assignment Vs Initialization

```
1  std::string get_value(); // A
2
3  void use_value() {
4    std::string s;   //  default construct
5    s = get_value(); //  move-assignment
6  }
```

https://godbolt.org/z/tqWjiJ

```
1  const std::string get_value(); // B
2
3  void use_value() {
4    std::string s;   //  default construct
5    s = get_value(); //  copy-assignment
6  }
```

https://godbolt.org/z/1tH2zd

```
1  std::string get_value(); // C
2
3  void use_value() {
4    const std::string s = get_value(); /// direct-init/RVO
5  }
```

https://godbolt.org/z/1ec2gI

```
1  const std::string get_value(); // D
2
3  void use_value() {
4    std::string s = get_value(); /// direct-init/RVO
5  }
```

https://godbolt.org/z/TFFieN

Reassignment

```
1  void use_value(int count) { // A
2    std::string val;
3    for (int i = 0; i < count; ++i) {
4      val = get_value();
5    }
6  }
```

https://godbolt.org/z/Hzu07y

```
1  void use_value(int count) { // B
2    for (int i = 0; i < count; ++i) {
3      std::string val = get_value();
4    }
5  }
```

https://godbolt.org/z/J758X4

Reassignment

```
1  void use_value(int count) { // A
2    std::string val; //  default construct
3    for (int i = 0; i < count; ++i) {
4      val = get_value(); //  copy/move assignment
5    }
6  }
```

https://godbolt.org/z/uh5zkL

```
1  void use_value(int count) { // B
2    for (int i = 0; i < count; ++i) {
3      std::string val = get_value(); /// direct-init/RVO
4    }
5  }
```

https://godbolt.org/z/d1EOVp

Triviality

`is_trivially_destructible`

[meta.unary.prop]

> *`is_destructible_v<T>` is `true` and `remove_all_extents_t<T>` is either a non-class type or a class type with a trivial destructor.*

is_trivially_destructible

```
1  #include <type_traits>
2
3  struct S {
4  };
```

is_trivially_destructible

```cpp
#include <type_traits>

struct S {
};
```

```cpp
static_assert(std::is_trivially_destructible_v<S>);
```

is_trivially_destructible

```cpp
#include <type_traits>

struct S {
  ~S() = default;
};
```

https://godbolt.org/z/51atoz

is_trivially_destructible

```cpp
#include <type_traits>

struct S {
  ~S() = default;
};
```

https://godbolt.org/z/51atoz

```cpp
static_assert(std::is_trivially_destructible_v<S>);
```

is_trivially_destructible

```cpp
#include <type_traits>

struct S {
  ~S() {}
};
```

https://godbolt.org/z/BVEyOv

is_trivially_destructible

```cpp
#include <type_traits>

struct S {
  ~S() {}
};
```

https://godbolt.org/z/BVEyOv

```cpp
static_assert(!std::is_trivially_destructible_v<S>);
```

is_trivially_destructible

```cpp
#include <type_traits>
#include <string>

struct S {
  std::string s;
};
```

https://godbolt.org/z/iJ_UV5

is_trivially_destructible

```cpp
#include <type_traits>
#include <string>

struct S {
  std::string s;
};
```

https://godbolt.org/z/iJ_UV5

```cpp
static_assert(!std::is_trivially_destructible_v<S>);
```

`is_trivially_copyable`

[meta.unary.prop]

> *T is a trivially copyable type*

[basic.types]

> *… the underlying bytes (6.6.1) making up the object can be copied into an array of* `char`*,* `unsigned char`*, or* `std::byte` *(17.2.1). If the content of that array is copied back into the object, the object shall subsequently hold its original value.*

`is_trivially_copyable`

[class.props]

A trivially copyable class is a class:

- *(1.1) that has at least one eligible copy constructor, move constructor, copy assignment operator, or move assignment operator*
- *(1.2) where each eligible copy constructor, move constructor, copy assignment operator, and move assignment operator is trivial, and*
- *(1.3) that has a trivial, non-deleted destructor*

is_trivially_copyable

```cpp
#include <type_traits>
#include <string>

static_assert(std::is_trivially_copyable_v<int>);

struct S {
  int i;
};
```

https://godbolt.org/z/ICmyLH

is_trivially_copyable

```cpp
#include <type_traits>
#include <string>

static_assert(std::is_trivially_copyable_v<int>);

struct S {
  int i;
};
```

https://godbolt.org/z/ICmyLH

```cpp
static_assert(std::is_trivially_copyable_v<S>);

static_assert(!std::is_trivially_copyable_v<std::string>);
```

is_trivially_copyable

```cpp
#include <type_traits>

struct S {
  ~S() {} /// not trivially destructible
};
```

https://godbolt.org/z/m-i17d

is_trivially_copyable

```cpp
#include <type_traits>

struct S {
  ~S() {} /// not trivially destructible
};
```

https://godbolt.org/z/m-i17d

No, non-trivially-destructible types are also non-trivially-copyable.

```cpp
static_assert(!std::is_trivially_copyable_v<S>);
```

`is_trivially_constructible`

[meta.unary.prop]

> *`is_constructible_v<T, Args...>` is `true` and the variable definition for `is_constructible`, as defined below, is known to call no operation that is not trivial*

is_trivially_constructible

```cpp
#include <type_traits>

struct S {
  int i;
  S() = default;
  S(int) {}
};

// no parameter constructor (default)
static_assert(std::is_trivially_constructible_v<S>);

// copy constructor
static_assert(std::is_trivially_constructible_v<S, const S &>);

// non-existent constructor
static_assert(!std::is_trivially_constructible_v<S, int, int >);

// User defined operation cannot be trivial
static_assert(!std::is_trivially_constructible_v<S, int>);
```

https://godbolt.org/z/foQeP5

`is_trivially_default_constructible`

[meta.unary.prop]

`is_trivially_constructible_v<T>` is `true`.

is_trivially_default_constructible

```cpp
#include <type_traits>

struct S {
  S() = default;
};
```

https://godbolt.org/z/llA9a3

is_trivially_default_constructible

```cpp
#include <type_traits>

struct S {
  S() = default;
};
```

https://godbolt.org/z/llA9a3

```cpp
// these are equiv
static_assert(std::is_trivially_constructible_v<S>);
static_assert(std::is_trivially_default_constructible_v<S>);
```

is_trivially_default_constructible

Is this trivially default constructible?

```cpp
#include <type_traits>

struct S {
  S() {}
};
```

https://godbolt.org/z/NuP-My

is_trivially_default_constructible

No, it has a user defined default constructor.

```
1  #include <type_traits>
2
3  struct S {
4    S() {}
5  };
6
7  static_assert(!std::is_trivially_default_constructible_v<S>);
```

https://godbolt.org/z/tcAoPJ

is_trivially_default_constructible

Is this trivially default constructible?

```cpp
#include <type_traits>

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

https://godbolt.org/z/IQjcs7

`is_trivially_default_constructible`

No, the default constructor does something: initializes `i` to `0`.

```
1  #include <type_traits>
2
3  struct S {
4    int i{};
5  };
6
7  static_assert(!std::is_trivially_default_constructible_v<S>);
```

https://godbolt.org/z/iYs7Pk

is_trivially_default_constructible

Without a default in class initializer it is:

```
1  #include <type_traits>
2
3  struct S {
4    int i; ///
5  };
6
7  static_assert(std::is_trivially_default_constructible_v<S>);
```

https://godbolt.org/z/T9DNhv

`is_trivially_copy_constructible`

[meta.unary.prop]

For a referenceable type T, the same result as `is_trivially_constructible_v<T, const T&>`*, otherwise* `false`*.*

is_trivially_copy_constructible

```cpp
#include <type_traits>

struct S {
  int i;
};

// equiv
static_assert(std::is_trivially_constructible_v<S, const S &>);
static_assert(std::is_trivially_copy_constructible_v<S>); https://godbolt.org/z/LC71KD
```

is_trivially_copy_constructible

Is this trivially copy constructible?

```cpp
#include <string>
#include <type_traits>

struct S {
  std::string s;
};
```

https://godbolt.org/z/67yKDH

`is_trivially_copy_constructible`

No, because `std::string` is not.

```
1  #include <string>
2  #include <type_traits>
3
4  struct S {
5    std::string s;
6  };
7
8  static_assert(!std::is_trivially_copy_constructible_v<S>);https://godbolt.org/z/C6YnzL
```

is_trivially_copy_constructible

Is this trivially copy constructible?

```cpp
#include <type_traits>

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

https://godbolt.org/z/IQjcs7

is_trivially_copy_constructible

Yes, trivial default constructibility does not affect trivial copy constructibility.

```cpp
#include <type_traits>

struct S {
  int i{};
};

static_assert(std::is_trivially_copy_constructible_v<S>);
static_assert(!std::is_trivially_default_constructible_v<S>);
```

https://godbolt.org/z/GORYEn

`is_trivially_move_constructible`

[meta.unary.prop]

> *For a referenceable type T, the same result as* `is_trivially_constructible_v<T, T&&>`*, otherwise* `false`*.*

is_trivially_move_constructible

```cpp
#include <type_traits>

struct S {
  int i;
};

// equiv
static_assert(std::is_trivially_constructible_v<S, S &&>);
static_assert(std::is_trivially_move_constructible_v<S>);
```

https://godbolt.org/z/525jmj

is_trivially_move_constructible

Is this trivially move constructible?

```cpp
#include <string>
#include <type_traits>

struct S {
  std::string s;
};
```

https://godbolt.org/z/67yKDH

`is_trivially_move_constructible`

No, because `std::string` is not.

```
1  #include <string>
2  #include <type_traits>
3
4  struct S {
5    std::string s;
6  };
7
8  static_assert(!std::is_trivially_move_constructible_v<S>);https://godbolt.org/z/n4K4Hv
```

`is_trivially_assignable`

[meta.unary.prop]

> *`is_assignable_v<T, U>` is `true` and the assignment, as defined by `is_assignable`, is known to call no operation that is not trivial*

is_trivially_assignable

```cpp
#include <string>
#include <type_traits>

struct S
{
  int i;

  S &operator=(const int a_i){ i = a_i; return *this; }
};
```

https://godbolt.org/z/LSSPHD

Is this trivially assignable, and in what ways?

is_trivially_assignable

```cpp
#include <string>
#include <type_traits>

struct S
{
  int i;

  S &operator=(const int a_i){ i = a_i; return *this; }
};

// copy assignment
static_assert(std::is_trivially_assignable_v<S, const S &>);

// non-existent assignment
static_assert(!std::is_trivially_assignable_v<S, std::string>);

// user-defined assignment operation not trivial
static_assert(!std::is_trivially_assignable_v<S, int>);
```

https://godbolt.org/z/-9n2FH

`is_trivially_copy_assignable`

[meta.unary.prop]

For a referenceable type T, the same result as `is_trivially_assignable_v<T&, const T&>`*, otherwise* `false`.

is_trivially_copy_assignable

```cpp
#include <type_traits>

struct S {
  int i;
};

// equiv
static_assert(std::is_trivially_assignable_v<S, const S &>);
static_assert(std::is_trivially_copy_assignable_v<S>);
```

https://godbolt.org/z/NuCP0y

is_trivially_copy_assignable

Is this trivially copy assignable?

```cpp
#include <string>
#include <type_traits>

struct S {
  std::string s;
};
```

https://godbolt.org/z/67yKDH

`is_trivially_copy_assignable`

No, because `std::string` is not.

```cpp
#include <string>
#include <type_traits>

struct S {
  std::string s;
};

static_assert(!std::is_trivially_copy_assignable_v<S>);
```

https://godbolt.org/z/EDjRDV

`is_trivially_move_assignable`

[meta.unary.prop]

For a referenceable type T, the same result as `is_trivially_assignable_v<T&, T&&>`*, otherwise* `false`*.*

is_trivially_move_assignable

```cpp
#include <type_traits>

struct S {
  int i;
};

// equiv
static_assert(std::is_trivially_assignable_v<S, S &&>);
static_assert(std::is_trivially_move_assignable_v<S>);
```

https://godbolt.org/z/IeHt3s

is_trivially_move_assignable

Is this trivially move assignable?

```cpp
#include <string>
#include <type_traits>

struct S {
  std::string s;
};
```

https://godbolt.org/z/67yKDH

`is_trivially_move_assignable`

No, because `std::string` is not.

```cpp
#include <string>
#include <type_traits>

struct S {
  std::string s;
};

static_assert(!std::is_trivially_move_assignable_v<S>);
```

https://godbolt.org/z/bRJc_c

`is_trivial`

[meta.unary.prop]

> *T is a trivial type*

[basic.types]

> *Scalar types, trivial class types, arrays of such types and cv-qualified versions of these types are collectively called trivial types.*

is_trivial

[class.props]

> *A trivial class is a class that is trivially copyable and has one or more eligible default constructors, all of which are trivial. [Note: In particular, a trivially copyable or trivial class does not have virtual functions or virtual base classes.—end note]*

is_trivial

```cpp
#include <type_traits>

struct S {
  int i;
};

static_assert(std::is_trivially_default_constructible_v<S>);
static_assert(std::is_trivially_destructible_v<S>);
static_assert(std::is_trivial_v<S>);
```

https://godbolt.org/z/dp2TAv

is_trivial

```cpp
#include <type_traits>

struct S {
  int i{};
};

static_assert(!std::is_trivially_default_constructible_v<S>);
static_assert(std::is_trivially_destructible_v<S>);
static_assert(!std::is_trivial_v<S>);
```

https://godbolt.org/z/sE7jnt

is_trivial

Is this trivial?

```
1  #include <type_traits>
2
3  struct S {
4    int i;
5    virtual void do_stuff(){}
6  };
```

https://godbolt.org/z/h9sC4P

is_trivial

No, the existence of a `virtual` function prevents it.

```cpp
#include <type_traits>

struct S {
  int i;
  virtual void do_stuff(){}
};

static_assert(!std::is_trivially_default_constructible_v<S>);
static_assert(std::is_trivially_destructible_v<S>);
static_assert(!std::is_trivial_v<S>);
```

https://godbolt.org/z/_xs5fg

Trivial Efficiency

Return int by value / move

```
1  int get_val() { // A
2    int val{5};
3    return std::move(val);
4  }
```
https://godbolt.org/z/RiPQGi

```
1  int get_val() { // B
2    int val{5};
3    return val;
4  }
```
https://godbolt.org/z/otcwQP

```
1  int get_val() { // C
2    const int val{5};
3    return std::move(val);
4  }
```
https://godbolt.org/z/qWS6VU

```
1  int get_val() { // D
2    const int val{5};
3    return val;
4  }
```
https://godbolt.org/z/luCviW

Return int by value / move

Irrelevant?, `int` is trivially copyable and moveable.

Return int by value / move

What happens when we disable optimizations?

```
1  #include <utility>
2
3  int get_value()
4  {
5    const int val = 5;
6    return std::move(val);
7  }
```

https://godbolt.org/z/MMnxiI

Return int by value / move

Or without the move?

```cpp
#include <utility>

int get_value()
{
  const int val = 5;
  return val;
}
```

https://godbolt.org/z/t2yiph

Return int by value / move

```cpp
int get_val() { // A
  auto [int1, int2] = get_int_pair();
  return std::move(int1);
}
```

https://godbolt.org/z/LpDeJ5

```cpp
int get_val() { // B
  auto [int1, int2] = get_int_pair();
  return int1;
}
```

https://godbolt.org/z/Pj3JoE

```cpp
int get_val() { // C
  const auto [int1, int2] = get_int_pair();
  return std::move(int1);
}
```

https://godbolt.org/z/nLxtuM

```cpp
int get_val() { // D
  const auto [int1, int2] = get_int_pair();
  return int1;
}
```

https://godbolt.org/z/NRHIR2

Return int by value / move

Irrelevant*, `int` is trivially copyable and moveable.

Pass By Value / Move

```
1  void do_things() { // A
2    int val{5};
3    use_int(val);
4  }
```

https://godbolt.org/z/aycN2t

```
1  void do_things() { // B
2    int val{5};
3    use_int(std::move(val));
4  }
```

https://godbolt.org/z/A0QObg

```
1  void do_things() { // C
2    use_int(5};
3  }
```

Pass By Value / Move

Irrelevant*, `int` is trivially copyable and moveable.

Out Parameter vs Return Value

```
1  void use_value() { // A
2    int s = get_value();
3  }
```

```
1  void use_value() { // B
2    const int s = get_value();
3  }
```

```
1  void use_value() { //C
2    int s;
3    get_value(s);
4  }
```

https://godbolt.org/z/mzqQDf

Out Parameter vs Return Value

Largely irrelevant*, all versions take 1 parameter: namely the memory address of the result.

Out Parameter vs Return Value

```
1  void use_value() {
2    int s{}; /// does this change things?
3    get_value(s);
4  }
```

https://godbolt.org/z/GdxcMg

Out Parameter vs Return Value

```
1  void use_value() {
2    int s{}; /// does this change things?
3    get_value(s);
4  }
```

https://godbolt.org/z/GdxcMg

Yes, it must now initialize the value before passing a reference to it.

Out Parameter vs Return Value

```
1  void use_value() {
2    int s;
3    get_value(s);
4  }
```

https://godbolt.org/z/C1qY02

Wait though, this is looking more suspicious the more I look at it…

Out Parameter vs Return Value

```
1  // Hypothetical implementation of get_value()
2  void get_value(int &val)
3  {
4    val += 10;
5  }
6
7  void use_value() {
8    int s;
9    get_value(s);
10 }
```

https://godbolt.org/z/6L7RaP

Assignment Vs Initialization

```
1  int get_value(); // A
2
3  void use_value() {
4    int s;
5    s = get_value();
6  }
```

https://godbolt.org/z/qi6fgG

```
1  const int get_value(); // B
2
3  void use_value() {
4    int s;
5    s = get_value();
6  }
```

https://godbolt.org/z/Q6MIPB

```
1  int get_value(); // C
2
3  void use_value() {
4    const int s = get_value();
5  }
```

https://godbolt.org/z/Z3Nwj7

```
1  const int get_value(); // D
2
3  void use_value() {
4    int s = get_value();
5  }
```

https://godbolt.org/z/4do0pd

Assignment Vs Initialization

Irrelevant*, `int` is trivially default constructible, trivially assignable.

Assignment Vs Initialization

```
1  const int get_value();
2
3  void use_value() {
4    int s{}; /// does this change things?
5    s = get_value();
6  }
```

https://godbolt.org/z/Dzx6ap

Assignment Vs Initialization

```
1  const int get_value();
2
3  void use_value() {
4    int s{}; /// does this change things?
5    s = get_value();
6  }
```

https://godbolt.org/z/Dzx6ap

Probably not, the compiler will optimize away the dead store.

Reassignment

```
1  void use_value(int count) { // A
2    int val;
3    for (int i = 0; i < count; ++i) {
4      val = get_value();
5    }
6  }
```

https://godbolt.org/z/cQ50EZ

```
1  void use_value(int count) { // B
2    for (int i = 0; i < count; ++i) {
3      const int val = get_value();
4    }
5  }
```

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

Assignment Vs Initialization

Irrelevant*, `int` is trivially default constructible, trivially assignable.

On *Irrelevant

With the optimizer disabled, the compiler still has to contend with things like

- setting up local stack space
- calling `std::move`

even when the types are trivial. The Best Practices options are still the best options.

Trivial Efficiency

Is `std::array<>` trivial?

Trivial Efficiency

Is `std::array<>` trivial?

If the contained type is, yes!

How Far Does Trivial Efficiency Go?

```cpp
#include <array>
struct S { std::array<int, 10> data; };

S get_S_good(bool value) {
  if (value) {
    return S{1,2,3,1,2,3,3,2,1,1024};
  } else {
    return S{1024,1,2,3,1,2,3,3,2,1};
  }
}

S get_S_bad(bool value) {
  const S opt1{1,2,3,1,2,3,3,2,1,1024};
  const S opt2{1024,1,2,3,1,2,3,3,2,1};

  if (value) {
    return opt1;
  } else {
    return opt2;
  }
}
```

https://godbolt.org/z/CtzF5-

Can Everything Be `trivial`?

Can Everything Be `trivial`?

Not everything, but most things. We can still make a runtime sized container…

```
1 #include <cstddef>
2
3 template<typename Value_Type, std::size_t Capacity>
4 struct Container;
```

Can Everything Be `trivial`?

Add something to hold the data:

```cpp
#include <array>
#include <cstddef>

template<typename Value_Type, std::size_t Capacity>
struct Container {
  std::array<Value_Type, Capacity> data;
};
```

https://godbolt.org/z/oRPfbM

Can Everything Be `trivial`?

Add the current size:

```cpp
#include <array>
#include <cstddef>

template<typename Value_Type, std::size_t Capacity>
struct Container {
  std::array<Value_Type, Capacity> data;
  std::size_t size{0};
};
```

https://godbolt.org/z/h07jCI

Can Everything Be `trivial`?

Add the ability to add data:

```cpp
#include <array>
#include <cstddef>

template<typename Value_Type, std::size_t Capacity>
struct Container {
  std::array<Value_Type, Capacity> data;
  std::size_t size{0};

  constexpr void push_back(Value_Type vt) {
    data[size++] = vt;
  }
};
```

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

Can Everything Be `trivial`?

What issues do we see?

```
#include <array>
#include <cstddef>

template<typename Value_Type, std::size_t Capacity>
struct Container {
  std::array<Value_Type, Capacity> data;
  std::size_t size{0};

  constexpr void push_back(Value_Type vt) {
    data[size++] = vt;
  }
};
```

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

Can Everything Be `trivial`?

What issues do we see?

```
1  #include <array>
2  #include <cstddef>
3
4  template<typename Value_Type, std::size_t Capacity>
5  struct Container {
6    std::array<Value_Type, Capacity> data;
7    std::size_t size{0};
8
9    constexpr void push_back(Value_Type vt) {
10     if (size == Capacity) { throw std::logic_error{"over capacity"}; } ///
11     data[size++] = vt;
12   }
13   /// to-do: all the other container things
14
15   // maybe this? or is it too restrictive?
16   // does it matter? The main issues are:
17   // default construction, erasing elements
18   static_assert(std::is_trivial_v<Value_Type>);
19 };
```

https://godbolt.org/z/bXGc_U

Can Everything Be `trivial`?

```
1  #include <array>
2  #include <cstddef>
3
4  template<typename Value_Type, std::size_t Capacity>
5  struct Container {
6    std::array<Value_Type, Capacity> data;
7    std::size_t size{0};
8    constexpr void push_back(Value_Type vt) {
9      if (size == Capacity) { throw std::logic_error{"over capacity"}; } ///
10     data[size++] = vt;
11   }
12   static_assert(std::is_trivial_v<Value_Type>);
13 };
14
15 int main() {
16   Container<int, 1000> c;
17   c.push_back(1);
18   c.push_back(2);
19   c.push_back(3);
20 }
```

https://godbolt.org/z/_YtLaW

Can Everything Be `trivial`?

Compared to:

```cpp
#include <vector>

int main() {
  std::vector<int> c;
  c.push_back(1);
  c.push_back(2);
  c.push_back(3);
}
```

https://godbolt.org/z/ylFHne

Can Everything Be `trivial`?

Or:

```cpp
#include <string>

int main() {
  std::basic_string<int> c;
  c.push_back(1);
  c.push_back(2);
  c.push_back(3);
}
```

https://godbolt.org/z/tbJDgl

`basic_string` is an interesting container in that it requires that any contained type be trivial.

Rule of Zero

The Rule of Zero

- Never define any of the special functions
- With class initializers this can even include the default constructor
- Likely will result in less code and smaller / more efficient compiled code

Great C++ `is_trivial`

Great C++ `is_trivial`

Great C++ `is_trivial`

- Being trivially destructible and trivially copyable is probably the best thing you can do to write optimizable code

Great C++ `is_trivial`

- Being trivially destructible and trivially copyable is probably the best thing you can do to write optimizable code
- Trivial default constructibility isn’t our main concern, as it doesn’t affect the other operations

Great C++ `is_trivial`

- Being trivially destructible and trivially copyable is probably the best thing you can do to write optimizable code
- Trivial default constructibility isn't our main concern, as it doesn't affect the other operations
- Follow the Rule of Zero

Great C++ `is_trivial`

- Being trivially destructible and trivially copyable is probably the best thing you can do to write optimizable code
- Trivial default constructibility isn’t our main concern, as it doesn’t affect the other operations
- Follow the Rule of Zero
- Triviality is one of the main keys for writing code that is `cosntexpr` friendly

Jason Turner

- Co-host of CppCast https://cppcast.com
- Host of C++ Weekly https://www.youtube.com/c/JasonTurner-lefticus
- Projects
  - https://chaiscript.com
  - https://cppbestpractices.com
  - https://github.com/lefticus/cpp_box
  - https://coloradoplusplus.info
- Microsoft MVP for C++ 2015-present

Jason Turner

Independent and available for training or contracting

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

Check out the “North Denver Metro C++ Meetup,” we’ve been meeting consistently since November 2016!