# Practical Performance Practices - Revisited

 @lefticus

# 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

 @lefticus

# About my Talks

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

 @lefticus

# Upcoming Events

- C++ On Sea 2019 Workshop Post Conference - “Applied `constexpr`” - 1 Day - Still available
- Core C++ 2019 Pre Conference Workshop - “Understanding Object Lifetime” - 1 Day

 

# Practical Performance Practices - Revisited

 @lefticus

# Background

 @lefticus

# Background

This is an update to a presentation I gave at C++Now 2016 called “Practical Performance Practices”

The premise is: ChaiScript was hard to optimize due to no particular hotspots in the code, so these are the guidelines I developed to ensure code that generally performs well.

We will cover what was then and what is today.

This version is either too long, or too short… we’ll see what happens.

 

# Are We Going Off In The Weeds?

Some of this might be a bit nitpicky about C++

but

The beauty of C++ is that we can argue and reason about these things!

 @lefticus

# Optimizing Compilers Are Amazing

 @lefticus

# Optimizing Compilers Are Amazing

```
1  #include <string>
2
3  int main()
4  {
5    std::string s("a");
6    return s.size();
7  }
```

 @lefticus

# Then - g++ 5.1+

```
1  main:
2          mov     eax, 1
3          ret
```

The first time I showed this I got many questions during and after the talk about how I got the assembly output of the program…

 

# Optimizing Compilers Are Amazing

```
1  #include <string>
2
3  int main()
4  {
5    return std::string("a").size() + std::string("b").size();
6  }
```

 @lefticus

# Then - g++ 5.1

```
.LC0:
        .string "basic_string::_M_construct null not valid"
void std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>
>::_M_construct<char const*>(char const*, char const*, std::forward_iterator_tag)
[clone .isra.14]:
        push    r12
        push    rbp
        mov     r12, rsi
        push    rbx
        mov     rbp, rdi
        sub     rsp, 16
        test    rsi, rsi
        jne     .L4
        test    rdx, rdx
        je      .L4
        mov     edi, OFFSET FLAT:.LC0
        call    std::__throw_logic_error(char const*)
.L4:
        mov     rbx, rdx
        sub     rbx, r12
        cmp     rbx, 15
        mov     QWORD PTR [rsp+8], rbx
        ja      .L17
        cmp     rbx, 1
        mov     rdi, QWORD PTR [rbp+0]
```

 

# Then

```
1           jne     .L5
2           movzx   eax, BYTE PTR [r12]
3           mov     BYTE PTR [rdi], al
4           jmp     .L6
5
6   .L17:
7           lea     rsi, [rsp+8]
8           xor     edx, edx
9           mov     rdi, rbp
10          call    std::__cxx11::basic_string<char, std::char_traits<char>,
11  std::allocator<char> >::_M_create(unsigned long&, unsigned long)
12          mov     rdx, QWORD PTR [rsp+8]
13          mov     QWORD PTR [rbp+0], rax
14          mov     rdi, rax
15          mov     QWORD PTR [rbp+16], rdx
16  .L5:
17          mov     rdx, rbx
18          mov     rsi, r12
            call    memcpy
```

 

# Then

```
1  .L6:
2          mov     rax, QWORD PTR [rsp+8]
3          mov     rdx, QWORD PTR [rbp+0]
4          mov     QWORD PTR [rbp+8], rax
5          mov     BYTE PTR [rdx+rax], 0
6          add     rsp, 16
7          pop     rbx
8          pop     rbp
9          pop     r12
10         ret
11 .LC2:
12         .string "a"
13 .LC3:
14         .string "b"
15 main:
16         push    rbx
17         mov     edx, OFFSET FLAT:.LC2+1
18         mov     esi, OFFSET FLAT:.LC2
```

 @lefticus

# Then

```
 1  sub     rsp, 64
 2  lea     rax, [rsp+16]
 3  mov     rdi, rsp
 4  mov     QWORD PTR [rsp], rax
 5  call    void std::__cxx11::basic_string<char, std::char_traits<char>,
 6  std::allocator<char> >::_M_construct<char const*>(char const*, char const*,
 7  std::forward_iterator_tag) [clone .isra.14]
 8  lea     rax, [rsp+48]
 9  lea     rdi, [rsp+32]
10  mov     edx, OFFSET FLAT:.LC3+1
11  mov     esi, OFFSET FLAT:.LC3
12  mov     rbx, QWORD PTR [rsp+8]
13  mov     QWORD PTR [rsp+32], rax
14  call    void std::__cxx11::basic_string<char, std::char_traits<char>,
15  std::allocator<char> >::_M_construct<char const*>(char const*, char const*,
16  std::forward_iterator_tag) [clone .isra.14]
17  mov     rdi, QWORD PTR [rsp+32]
18  lea     rax, [rsp+48]
    add     ebx, DWORD PTR [rsp+40]
    cmp     rdi, rax
    je      .L19
    call    operator delete(void*)
```

 

# Then

```
1  .L19:
2          mov     rdi, QWORD PTR [rsp]
3          lea     rax, [rsp+16]
4          cmp     rdi, rax
5          je      .L24
6          call    operator delete(void*)
7  .L24:
8          add     rsp, 64
9          mov     eax, ebx
10         pop     rbx
11         ret
12         mov     rdi, QWORD PTR [rsp]
13         lea     rdx, [rsp+16]
14         mov     rbx, rax
15         cmp     rdi, rdx
16         je      .L22
17         call    operator delete(void*)
18 .L22:
19         mov     rdi, rbx
20         call    _Unwind_Resume
```

 

# Optimizing Compilers Are Amazing

- But trying to predict what the compiler can optimize is a risky game

# But It's Been 3 Years, Have Things Changed?

 @lefticus

# Optimizing Compilers Are Amazing - 2019

```
1  #include <string>
2
3  int main()
4  {
5    return std::string("a").size() + std::string("b").size();
6  }
```

 @lefticus

# Optimizing Compilers Are Amazing - 2019

And compilers differ…

```
1  #include <string>
2
3  int main()
4  {
5    return std::string("a").size() + std::string("b").size()
6         + std::string("c").size();
7  }
```

# Optimizing Compilers Are Amazing - 2019

- But trying to predict what the compiler can optimize is still a risky game

# Performance Practices

 @lefticus

# Which Is Better In Normal Use?

`std::vector`

- or -

`std::list`

- WHY?

 

# std::list

 @lefticus

# std::list

```
1  #include <list>
2
3  int main()
4  {
5    std::list<int> v{1, 2};
6  }
```

- Allocate a new node
- Handle exception thrown during node allocation?
- Assign the value
- Hook up some pointers
- Do it all over again
- Delete node
- Delete node

 

# `std::vector`

 @lefticus

# std::vector

```
1  #include <vector>
2
3  int main()
4  {
5    std::vector<int> v{1, 2};
6  }
```

- Allocate a buffer
- Assign a value in the buffer
- Delete the buffer

 @lefticus

# What about `std::array`?

```
1  #include <array>
2
3  int main()
4  {
5    std::array<int, 2> a{1, 2};
6  }
```

 @lefticus

# What about `std::array`?

- Code is completely compiled away

 

# C++17 Makes The `std::array` Use Case Better

```
1  #include <array>
2
3  int main()
4  {
5    std::array a{1, 2};
6  }
```

 

# And What Else Does `std::array` Give Us?

```
1  #include <array>
2
3  int main()
4  {
5    constexpr std::array a{1, 2};
6  }
```

 

# And What About `vector`/`list` With libc++?

```
1  #include <array>
2  #include <vector>
3  #include <list>
4
5  int main()
6  {
7    std::array a{1, 2};
8    std::vector v{1, 2};
9    std::list l{1}; // and if we have more than one?
10 }
```

Clang's heap elision: added with clang 3.1, overlooked by me in 2016.

 

# And Compared To `std::deque`?

```
1  #include <deque>
2
3  int main()
4  {
5    std::deque d{1, 2};
6  }
```

 @lefticus

# Container Practices

- Always prefer `std::array`
- Then `std::vector`
- Then only differ if you need specific behavior
- *And still measure*
- Make sure you understand what the library has to do

Questions?

 

# Don’t Do More Work Than You Have To

 @lefticus

# Don’t Do More Work Than You Have To

```
1  #include <string>
2
3  int main()
4  {
5    std::string s;
6    s = "A Somewhat Rather Long String";
7  }
```

- Construct a string object
- Reassign string object

 @lefticus

# *Always* `const`

```
1  #include <string>
2
3  int main()
4  {
5    const std::string s = "A Somewhat Rather Long String";
6  }
```

- Construct and initialize in one step
- At least ~32% more efficient
- Even more so with llvm's libc++

 @lefticus

# Question Not Asked in 2016: Why?

```
1  #include <string>
2
3  int main()
4  {
5    std::string s;
6    s = "A Somewhat Rather Long String";
7  }
```

What code does the compiler have to generate here?

 @lefticus

# Question Not Asked in 2016: Why?

```
1  #include <string>
2
3  int main()
4  {
5    std::string s;
6    s = "A Somewhat Rather Long String";
7  }
```

- Default constructor
- Assignment operator
  - Check for existing data, delete it
  - Copy in new data with new size
- stdlib (libc++) can have a huge impact
- But the simple thing is still the best thing.

 

# Always `const` - Complex Initialization

```
1  #include <string>
2  int main(const int argc, const char *[])
3  {
4    std::string s;
5    switch (argc % 4) {
6      case 0:
7        s = "long string is mod 0";
8        break;
9      case 1:
10       s = "long string is mod 1";
11       break;
12     case 2:
13       s = "long string is mod 2";
14       break;
15     case 3:
16       s = "long string is mod 3";
17       break;
18   }
19 }
```

- How can we make `s` `const` in this context?

 

# Always `const` - *Use IIFE*

```
 1  #include <string>
 2  int main(const int argc, const char *[])
 3  {
 4    const std::string s = [&](){ ///
 5      switch (argc % 4) {
 6        case 0:
 7          return "long string is mod 0";
 8        case 1:
 9          return "long string is mod 1";
10        case 2:
11          return "long string is mod 2";
12        case 3:
13          return "long string is mod 3";
14      }
15    }(); /// invoke
16  }
```

- ~31% more efficient
- But it turns out this is *accidentally* the most efficient version…

 

# Always `const` - *Use IIFE*

```
1  #include <string>
2  int main(const int argc, const char *[])
3  {
4    const std::string s = [&](){
5      switch (argc % 4) {
6        case 0:
7          return "long string is mod 0";
8        case 1:
9          return "long string is mod 1";
10       case 2:
11         return "long string is mod 2";
12       case 3:
13         return "long string is mod 3";
14     }
15   }();
16 }
```

- What is the return type of the lambda?
- What happens if one string is a different length?

 @lefticus

# Always `const` - *Use IIFE*

```
1  #include <string>
2  int main(const int argc, const char *[])
3  {
4    const std::string s = [&](){
5      switch (argc % 4) {
6        case 0:
7          return "long string is mod 0";
8        case 1:
9          return "long string is mod 1";
10       case 2:
11         return "long string is mod 2";
12       case 3:
13         return "a different length";
14     }
15   }();
16 }
```

- `strlen` must now be called to construct the `std::string` object

 

# Always `const` - *Use IIFE*

```
1  #include <string>
2  int main(const int argc, const char *[])
3  {
4    const std::string s = [&]() -> std::string {
5      switch (argc % 4) {
6        case 0:
7          return "long string is mod 0";
8        case 1:
9          return "long string is mod 1";
10       case 2:
11         return "long string is mod 2";
12       case 3:
13         return "a different length";
14     }
15   }();
16 }
```

- RVO now comes into play and no `strlen` needed

 

# Even With clang/libc++ Heap Elision These Hold (Do we go back and look?)

# *Always Initialize* When `const` Isn't Practical

 @lefticus

# *Always Initialize* When `const` Isn't Practical

```
1  struct Int
2  {
3    Int(std::string t_s)
4    {
5      m_s = t_s;
6    }
7
8    int val() const {
9      return std::atoi(m_s.c_str());
10   }
11
12   std::string m_s;
13 };
```

- Same issues as previous examples

 

# *Always Initialize* When `const` Isn't Practical

```
1  struct Int
2  {
3    Int(std::string t_s) : m_s(std::move(t_s))
4    {
5    }
6
7    int val() const {
8      return std::atoi(m_s.c_str());
9    }
10
11   std::string m_s;
12 };
```

- Same gains as const initializer
- What's wrong with this version now?
- `val()` parses string on each call

 

# Calculate on First Use?

```
1  struct Int {
2    Int(std::string t_s) : s(std::move(t_s)) { }
3
4    int val() const {
5      if (!is_calculated) {
6        value = std::atoi(s);
7      }
8      return value;
9    }
10
11   mutable bool is_calculated = false;
12   mutable int value;
13   std::string s;
14 };
```

- What's wrong now?
- C++ Core Guidelines state that const methods should be thread safe
- What else?
- `is_calculated` isn't being set

 

# Calculate On First Use?

```
1  struct Int {
2    Int(std::string t_s) : s(std::move(t_s)) { }
3
4    int val() const {
5      if (!is_calculated) {
6        value = std::atoi(s);
7        is_calculated = true;
8      }
9      return value;
10   }
11
12   mutable std::atomic_bool is_calculated = false;
13   mutable std::atomic_int value;
14   std::string s;
15 };
```

- Branching is slower
- Atomics are slower

 

# Calculate At Construction!

```
1  struct Int {
2    Int(const std::string &t_s) : m_i(std::atoi(t_s.c_str()))
3    { }
4
5    int val() const {
6      return m_i;
7    }
8
9    int m_i;
10 };
```

- No branching, no atomics, smaller runtime (`int` vs `string`)
- In the context of a large code base, this took ~2 years to find
- Resulted in 10% performance improvement across system
- *The simpler solution is almost always the best solution*

 @lefticus

# Initialization Practices

- Always const
- Always initialize
- Using IIFE can help you initialize
- Don’t recalculate values that can be calculated once
- *Reduce branching*

Questions?

# Don’t Do More Work Than You Have To

```
1  struct Base {
2    virtual ~Base() = default;
3    virtual void do_a_thing() = 0;
4  };
5
6  struct Derived : Base {
7    virtual ~Derived() = default;
8    void do_a_thing() override {}
9  };
```

- What’s wrong here?
- move construction / assignment is disabled (virtual destructor)
- `virtual ~Derived()` is unnecessary

 

# *Don’t Disable Move / Use Rule of 0/5*

```
1  struct Base {
2    virtual ~Base() = default;
3    Base() = default;
4    Base(const Base &) = default; Base& operator=(const Base&) = default;
5    Base(Base &&) = default; Base& operator=(Base &&) = default;
6
7    virtual void do_a_thing() = 0;
8  };
9
10 struct Derived : Base {
11   virtual void do_a_thing() {}
12 };
```

- 10% improvement with fixing this in just one commonly used class
- Annoying boilerplate, but keep it limited to the base class
- *Or disable copies and moves of polymorphic types in general?*

 

# On The Topic Of Copying

 @lefticus

# On The Topic Of Copying

```
#include <string>

struct S {
  S(std::string t_s) : s(std::move(t_s)) {}
  std::string s;
};

int main()
{
  for (int i = 0; i < 10000000; ++i) {
    std::string s = std::string("a not very short string") + "b";
    S o(s);
  }
}
```

- We all know that copying objects is bad
- So let's use `std::move`

 

# On The Topic Of Copying

```
#include <string>

struct S {
  S(std::string t_s) : s(std::move(t_s)) {}
  std::string s;
};

int main()
{
  for (int i = 0; i < 10000000; ++i) {
    std::string s = std::string("a not very short string") + "b";
    S o(std::move(s));
  }
}
```

- 29% more efficient
- 32% smaller binary
- *Why smaller binary?*
- Good! But what's better?

 

# *Avoid Named Temporaries*

```
1  #include <string>
2
3  struct S {
4    S(std::string t_s) : s(std::move(t_s)) {}
5    std::string s;
6  };
7
8  int main()
9  {
10   for (int i = 0; i < 10000000; ++i) {
11     S o(std::string("a not very short string") + "b");
12   }
13 }
```

- 2% more efficient again
- Can lead to less readable code sometimes, but more maintainable than `std::move` calls
- This is taking the “don’t declare a variable until you need it” philosophy to its ultimate conclusion

 @lefticus

# Avoid Copies

```
1  #include <memory>
2
3  int use_a_value(std::shared_ptr<const int> p)
4  {
5    return *p * 2;
6  }
7
8  int main()
9  {
10   auto ptr = std::make_shared<int>(42);
11   return use_a_value(ptr);
12 }
```

- What's the problem here?
- Copies are being made of `shared_ptr<int>`

# Avoid (`shared_ptr`) Copies

```
#include <memory>

int use_a_value(const std::shared_ptr<const int> &p)
{
  return *p * 2;
}

int main()
{
  auto ptr = std::make_shared<int>(42);
  use_a_value(ptr);
}
```

- Fixed!
- Right?
- Wrong!

 

# Avoid (`shared_ptr`) Copies

```
1  #include <memory>
2
3  int use_a_value(const std::shared_ptr<const int> &p) ///
4  {
5    return *p * 2;
6  }
7
8  int main()
9  {
10   auto ptr = std::make_shared<int>(42);
11   use_a_value(ptr);
12 }
```

Invoking implicit conversion from `shared_ptr<int>` to `shared_ptr<const int>`

 

# Avoid (`shared_ptr`) Copies

```
1  #include <memory>
2
3  int use_a_value(const std::shared_ptr<const int> &p)
4  {
5    if (p) {
6      return *p * 2;
7    } else {
8      return 0; ///?
9    }
10 }
11
12 int main()
13 {
14   auto ptr = std::make_shared<int>(42);
15   use_a_base(ptr);
16 }
```

 @lefticus

# *Avoid Automatic Conversions*

```
1  int use_a_value(const int p)
2  {
3    return p * 2;
4  }
5
6  int main()
7  {
8    auto ptr = std::make_shared<int>();
9    return use_a_value(*ptr);
10 }
```

- This version is *at least* 2.5x faster than the last

*Never pass a smart pointer unless you need to participate in the lifetime of the object*

 

# I Still Don't Like `std::endl`

 @lefticus

# `std::endl`

```
1  void println(ostream &os, const std::string &str)
2  {
3    os << str << std::endl;
4  }
```

- What does `std::endl` do?
- it's equivalent to `'\n' << std::flush`
- Expect that flush to cost you at least 9x overhead in your IO

 

# Real World `std::endl` Anecdote

```
1  void write_file(std::ostream &os) {
2    os << "a line of text" << std::endl;
3    os << "another line of text" << std::endl;
4    /* snip */
5    os << "many more lines of text" << std::endl;
6  }
7
8  void write_file(const std::string &filename) {
9    std::ofstream ofs(filename.c_str());
10   write_file(ofs);
11 }
12
13 std::string get_file_as_string() {
14   std::stringstream ss;
15   write_file(ss);
16   return ss.str();
17 }
```

 @lefticus

# *Avoid* `std::endl`

Prefer just using `'\n'`

```
1  void println(ostream &os, const std::string &str)
2  {
3    os << str << '\n';
4  }
```

 @lefticus

# What About `constexpr`?

 @lefticus

# What About `constexpr`?

```
1  #include <initializer_list>
2  template<typename Itr>
3  constexpr bool is_sorted(Itr begin, const Itr &end)
4  {
5    Itr start = begin;
6    ++begin;
7    while (begin != end) {
8      if (!(*start < *begin)) { return false; }
9      start = begin;
10     ++begin;
11   }
12   return true;
13 }
14
15 template<typename T>
16 constexpr bool is_sorted(const std::initializer_list<T> &l) {
17   return is_sorted(l.begin(), l.end());
18 }
19
20 int main()
21 {
22   return is_sorted({1,2,3,4,5});
23 }
```

 @lefticus

# What About *Not* `constexpr`?

```
1  #include <initializer_list>
2  template<typename Itr>
3  bool is_sorted(Itr begin, const Itr &end)
4  {
5    Itr start = begin;
6    ++begin;
7    while (begin != end) {
8      if (!(*start < *begin)) { return false; }
9      start = begin;
10     ++begin;
11   }
12   return true;
13 }
14
15 template<typename T>
16 bool is_sorted(const std::initializer_list<T> &l) {
17   return is_sorted(l.begin(), l.end());
18 }
19
20 int main()
21 {
22   return is_sorted({1,2,3,4,5});
23 }
```

- What does this compile to?

 @lefticus

# `constexpr`

- I use `constexpr` with care
- Full `constexpr` enabling of every data structure that can be can result in bigger code
- Bigger code is often slower code
- This is a profile and test scenario for me

 

# Wait, what!?

# That Was 2016

# Today?

# `constexpr` All The Things!

 @lefticus

# `constexpr` - 2019

Never do anything at runtime that you can do at compile time!

 @lefticus

# Modifying ChaiScript for `constexpr`

Net result of moving static data into `constexpr` data:

gcc:

- almost 20% better start up times
- very painful regression with function level `constexpr` objects
- 7.5% less memory usage during compile time

clang:

- ~10% better startup times
- 5% less memory usage during compile time

 @lefticus

# Modifying ChaiScript for `constexpr`

- Make sure all algorithms are enabled for `constexpr`
- Try to make all data structures `constexpr` capable
- Be aware that some compilers still need help
- See also “Practical `constexpr`” from Meeting C++ 2017
- Come to my class ;)

 @lefticus

# Triviality is Very Important

`constexpr` enabled types must still be `is_trivially_destructible`.

 @lefticus

# Review Time

```
1  struct S
2  {
3  };
```

Is S trivially destructible?

Yes.

 @lefticus

# Review Time

```
1  struct S
2  {
3    ~S() = default;
4  };
```

Is S trivially destructible?

Yes.

 

# Review Time

```
1  struct S
2  {
3    ~S() {}
4  };
```

Is `S` trivially destructible?

No.

 

# Review Time

```
1  struct S
2  {
3    ~S();
4  };
```

Is `S` trivially destructible?

No. It's impossible to make it `trivially_destructible` when forward declaring the destructor.

 

# Review Time

```
1  struct S
2  {
3    std::string s;
4  };
```

Is `S` trivially destructible?

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

But this code has an important distinction: the destructor can be inlined.
*Simply because we didn't get in the way.*

 @lefticus

# Hidden Work Practices

- Calculate values once - at initialization time
- Obey the rule of 0
- If it looks simpler, it's probably faster
- Avoid object copying
- Avoid automatic conversions
  - Don't pass smart pointers
  - Make conversion operations explicit
- Have IO performance problems? *Never use* `std::endl` *when writing to an unknown* `ostream`
- `constexpr` and `trivial` types are the ultimate in not doing more work than you have to

 @lefticus

# `shared_ptr`

 @lefticus

# `shared_ptr` Instantiations

```
1  #include <memory>
2
3  int main()
4  {
5    std::make_shared<int>(1);
6  }
```

- What does this have to do?

 

# `unique_ptr` Instantiations

```
1  #include <memory>
2
3  int main()
4  {
5    std::make_unique<int>(0);
6  }
```

- What does this have to do?
- Oh clang, you keep ruining my examples.

 @lefticus

# Compare To Manual Memory Management

```
1  int main()
2  {
3    auto i = new int(0);
4    delete i;
5  }
```

 @lefticus

# `shared_ptr`

- Possibly the biggest hammer in the post C++11 toolbox
- Should be used sparingly
- *2019 - Yes* `shared_ptr` *is a big hammer, but let's not underestimate* `std::map` *and* `std::deque`

 @lefticus

# Don’t Do More Work Than You Have To - Summary

 @lefticus

# Don’t Do More Work Than You Have To - Summary

- Avoid `shared_ptr`
- Avoid `std::endl`
- Always `const`
- Always initialize with meaningful values
- Don’t recalculate immutable results

Questions?

 

# Smaller Code Is Faster Code

 @lefticus

# Why Is Smaller Code Often Faster Code?

- RAM is slow
- Multiple Levels of faster cache is used to mitigate the performance problems
- The fastest L1 cache is also the smallest (it's expensive)
- I recently got a new Ryzen 5 2600 (2018 model). Who knows how much L1 instruction cache it has?

 

# Why Is Smaller Code Often Faster Code?

My L1 instruction cache (per core):

 @lefticus

# Why Is Smaller Code Often Faster Code?

- Programming C++ for the Commodore 64 isn’t just a neat trick
- If we can manage to get all code to fit in the I-cache we can see huge gains
- The entire program in 64k is probably unlikely, but local algorithms/loops/etc is quite doable

 

# Smaller Code Is Faster Code

```
1  struct B
2  {
3    virtual ~B() = default; // plus the other default operations
4    virtual std::vector<int> get_vec() const = 0;
5  };
6
7  template<typename T>
8  struct D : B
9  {
10   std::vector<int> get_vec() const override { return m_v; }
11   std::vector<int> m_v;
12 }
```

- With many template instantiations this code blows up in size quickly

 

# DRY In Templates

```
1  struct B
2  {
3    virtual ~B() = default; // plus the other default operations
4    virtual std::vector<int> get_vec() const { return m_v; }
5    std::vector<int> m_v;
6  };
7
8  template<typename T>
9  struct D : B
10 {
11 }
```

# Smaller Code Is Faster Code - Factories

 @lefticus

# Factories

```
1  struct B {
2    virtual ~B() = default;
3  };
4
5  template<int T>
6  struct D : B {
7  };
8
9  template<int T>
10 std::shared_ptr<B> d_factory() {
11   return std::make_shared<D<T>>();
12 }
13
14 int main() {
15   std::vector<std::shared_ptr<B>> v{
16     d_factory<1>(), d_factory<2>(), /* ... */ , d_factory<29>(), d_factory<30>()
17   };
18 }
```

- Prefer returning `unique_ptr<>` (Back To The Basics - Herb Sutter ~0:19)
- We already saw that `shared_ptr<>` is big - don't make more than you have to

 

# *Prefer return `unique_ptr<>` from factories*

```
1  struct B {
2    virtual ~B() = default;
3  };
4
5  template<int T>
6  struct D : B {
7  };
8
9  template<int T>
10 std::unique_ptr<B> d_factory() {
11   return std::make_unique<D<T>>();
12 }
13
14 int main() {
15   std::vector<std::shared_ptr<B>> v{
16     d_factory<1>(), d_factory<2>(), /* ... */ , d_factory<29>(), d_factory<30>()
17   };
18 }
```

 

# 2016 Data

```
1  template<int T> std::unique_ptr<B> d_factory()
2  {
3    return std::make_unique<D<T>>();
4  }
```

1.30s compile, 30k exe, 149796k compile RAM

```
1  template<int T> std::shared_ptr<B> d_factory()
2  {
3    return std::make_shared<D<T>>();
4  }
```

2.24s compile, 70k exe, 164808k compile RAM

```
1  template<int T> std::shared_ptr<B> d_factory()
2  {
3    return std::make_unique<D<T>>();
4  }
```

2.43s compile, 91k exe, 190044k compile RAM

 @lefticus

# (2019) GCC -O3

```
1  template<int T> std::unique_ptr<B> d_factory()
2  {
3    return std::make_unique<D<T>>();
4  }
```

3.5s compile, 35k exe, 117788k compile RAM

```
1  template<int T> std::shared_ptr<B> d_factory()
2  {
3    return std::make_shared<D<T>>();
4  }
```

3.87s compile, 75k exe, 115392k compile RAM

```
1  template<int T> std::shared_ptr<B> d_factory()
2  {
3    return std::make_unique<D<T>>();
4  }
```

5.83s compile, 112k exe, 132172k compile RAM

 

# (2019) clang libstdc++ -O3

```
1  template<int T> std::unique_ptr<B> d_factory()
2  {
3    return std::make_unique<D<T>>();
4  }
```

2.72s compile, 32k exe, 126872k compile RAM

```
1  template<int T> std::shared_ptr<B> d_factory()
2  {
3    return std::make_shared<D<T>>();
4  }
```

2.99s compile, 65k exe, 114300k compile RAM

```
1  template<int T> std::shared_ptr<B> d_factory()
2  {
3    return std::make_unique<D<T>>();
4  }
```

4.48s compile, 93k exe, 131816k compile RAM

 

# (2019) clang libc++ -O3

```
1  template<int T> std::unique_ptr<B> d_factory()
2  {
3    return std::make_unique<D<T>>();
4  }
```

2.03s compile, 31k exe, 109164k compile RAM

```
1  template<int T> std::shared_ptr<B> d_factory()
2  {
3    return std::make_shared<D<T>>();
4  }
```

2.84s compile, 59k exe, 123252k compile RAM

```
1  template<int T> std::shared_ptr<B> d_factory()
2  {
3    return std::make_unique<D<T>>();
4  }
```

2.86s compile, 71k exe, 122720k compile RAM

 

# *Prefer return `unique_ptr<>` from factories*

Our new compilers and standard library implementations are mitigating the differences.

 @lefticus

# A Note About Performance

```
1  template<int T> std::shared_ptr<B> d_factory()
2  {
3    return std::make_shared<D<T>>();
4  }
```

- This `make_shared` version is faster in raw performance
- If you create many short-lived shared objects, the `make_shared` version is fastest
- If you create long-lived shared objects, use the `make_unique` version is fastest
- C++ Core Guidelines are surprisingly inconsistent in examples for factories

 

# `std::function` and `std::bind`

 @lefticus

# Avoid `std::function<>`

```
1  #include <string>
2  #include <functional>
3
4  std::string add(const std::string &lhs, const std::string &rhs) {
5    return lhs + rhs;
6  }
7
8  int main() {
9    const std::function<std::string (const std::string &)> f
10     = std::bind(add, "Hello ", std::placeholders::_1);
11   f("World");
12 }
```

- 2.9x slower than bare function call
- 30% compile time overhead
- ~10% compile size overhead
- Held pretty steady for gcc, libc++ made some headway

 

# *Never. Ever. Ever. Use `std::bind`*

```
1  #include <string>
2  #include <functional>
3
4  std::string add(const std::string &lhs, const std::string &rhs) {
5    return lhs + rhs;
6  }
7
8  int main() {
9    const auto f = std::bind(add, "Hello ", std::placeholders::_1);
10   f("World");
11 }
```

- 1.9x slower than bare function call
- ~15% compile time overhead
- Effective Modern C++ #34
- Any talk on `std::function` from STL

 

# *Use Lambdas*

```
1  #include <string>
2
3  std::string add(const std::string &lhs, const std::string &rhs) {
4    return lhs + rhs;
5  }
6
7  int main() {
8    const auto f = [](const std::string &b) {
9      return add("Hello ", b);
10   };
11   f("World");
12 }
```

- 0 overhead compared to direct function call
- 0% compile time overhead

 

# 2019 - Optimizing Compilers Are More Amazing

```
#include <functional>

int add(const int lhs, const int rhs) {
  return lhs + rhs;
}

int main(const int argc, const char *[]) {
  const std::function<int (const int)> f
    = std::bind(add, 1, std::placeholders::_1);
  return f(2);
}
```

clang 4.0.0 (March 2017) can make this all go away.

 

# Smaller Code Is Faster Code - Exceptions

What happens with optimizers here? With various architectures.

```
1  #include <vector>
2  #include <algorithm>
3  #include <cstdint>
4
5  size_t mycount(const std::vector<uint8_t> &s, uint8_t c)
6  {
7    return std::count(std::begin(s), std::end(s), c);
8  }
```

 @lefticus

# Smaller Code Is Faster Code - Exceptions

- The compiler has unrolled and vectorized the loop for us
- So, you may see smaller/simpler code actually cause an increase in compile size
- Is this necessarily a good thing all the time?
- Every measurable code size decrease has resulted in a measurable performance increase for ChaiScript

 

# Smaller Code Is Faster Code - Summary

- Don’t repeat yourself in templates
- Avoid use of `shared_ptr`
- Avoid `std::function`
- Never use `std::bind`

Questions?

 @lefticus

# When I Break The Rules

 @lefticus

# `std::map`

- For very small, short lived key value pairs, std::vector can be faster
- Even if you are doing lots of querying of the keys

```
1 | std::vector<std::pair<std::string, int>> data;
```

vs

```
1 | std::map<std::string, int> data;
```

- This is similar to the `boost::flat_map`
- *But is this breaking a rule or following the other rules?*

 

# Avoid Non-Local Data

 @lefticus

# Non-Locals Tend To

1. Be statics - which have a cost associated
2. Need some kind of mutex protection
3. Be in a container with on-trivial lookup costs (`std::map<>` for example)

 @lefticus

# Let’s Chat About Statics

```
1  #include <string>
2
3  const std::string &get_string()
4  {
5    const static std::string val{"Hello World"}; ///
6    return val;
7  }
```

How do we do this better in C++17?

 @lefticus

# Let’s Chat About Statics

```
1  #include <string>
2
3  constexpr std::string_view get_string()
4  {
5    return "Hello World";
6  }
```

Maybe This?

 @lefticus

# Summary

 @lefticus

# Summary

- First ask yourself: What am I asking the compiler to do here?

 @lefticus

# Initialization Practices

- Always `const`
- Always initialize

 

# Hidden Work Practices

- Calculate values once - at initialization time
- Obey the rule of 0
- If it looks simpler, it’s probably faster
- Avoid automatic conversions - use `explicit`
- avoid `std::endl`

 

# Container Practices

- Always prefer `std::array`
- Then `std::vector`
- Then only differ if you need specific behavior
- Make sure you understand what the library has to do

 @lefticus

# Smaller Code Is Faster Code Practices

- Don’t repeat yourself in templates
- Avoid use of `std::shared_ptr`
- Avoid `std::function`
- Never use `std::bind`

 @lefticus

# Final Conclusions From 2019

- Always `const`
- Avoid the heap
- If there's a question, lambdas are the answer
- Utilize `constexpr`

 @lefticus

# So Why Does This All Work?

 @lefticus

# Branches and Predictions

- Code branches are expensive
- Simpler code has fewer branches

 

# Cache Hits

- CPU cache is many (hundreds of) times faster than main memory
- Smaller code (and simpler code is smaller) is more likely to fit in to the CPU cache

 

# Giving The Compiler As Much Info As Possible

- Let the compiler sort out the special member functions
- Use `constexpr`, `const`, and `noexcept` correctly

 @lefticus

# Doing What The Compiler Writer Expects

- Idiomatic C++ falls into certain patterns that compiler authors expect to find
- Well known patterns can be optimized better
- Compilers can remove branches from code it understands
- This is becoming all even more true as best-practice idioms are becoming more well established
- Chasing specific optimizations from compiler/stdlib implementations is fragile, but the simpler the more optimizable

 

# 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