CPPCON / 16TH SEPTEMBER 2020

JUST-IN-TIME COMPILATION: THE NEXT BIG THING?

BEN DEANE & KRIS JUSIAK - QUANTLAB FINANCIAL

AGENDA

AGENDA

- MOTIVATION

AGENDA

- MOTIVATION
- EXISTING SOLUTIONS

AGENDA

- MOTIVATION
- EXISTING SOLUTIONS
- HOW IT WORKS*

---

* BASED ON P1609R1

AGENDA

- MOTIVATION
- EXISTING SOLUTIONS
- HOW IT WORKS*
- EXAMPLES: C++20 AND BEYOND

---

* BASED ON P1609R1

AGENDA

- MOTIVATION
- EXISTING SOLUTIONS
- HOW IT WORKS*
- EXAMPLES: C++20 AND BEYOND
- SUMMARY / FUTURE?

---

* BASED ON P1609R1

MOTIVATION

WHAT IS JITTING?

WHAT IS JITTING?

- INSTEAD OF AHEAD-OF-TIME (AOT), AKA "THE NORM": IT'S LIKE THE APPLICATION (/LIBRARY) IS THE COMPILER...

WHAT IS JITTING?

- INSTEAD OF AHEAD-OF-TIME (AOT), AKA "THE NORM": IT'S LIKE THE APPLICATION (/LIBRARY) IS THE COMPILER...
- JUST-IN-TIME (JIT) IS COMPILATION AT THE POINT OF NEED

WHAT IS JITTING?

- INSTEAD OF AHEAD-OF-TIME (AOT), AKA "THE NORM": IT'S LIKE THE APPLICATION (/LIBRARY) IS THE COMPILER...
- JUST-IN-TIME (JIT) IS COMPILATION AT THE POINT OF NEED
- IT'S LIKE THE APPLICATION (/LIBRARY) IS THE COMPILER...

WHY JIT-COMPILE?

WHY JIT-COMPILE?

- AOT IS NOT POSSIBLE

WHY JIT-COMPILE?

- AOT IS NOT POSSIBLE
- AOT IS NOT DESIRABLE

WHY JIT-COMPILE?

- AOT IS NOT POSSIBLE
- AOT IS NOT DESIRABLE
- JIT GIVES PERF BENEFITS

WHY JIT-COMPILE?

- AOT IS NOT POSSIBLE
- AOT IS NOT DESIRABLE
- JIT GIVES PERF BENEFITS
- JIT GIVES PRODUCTIVITY BENEFITS

WHY JIT-COMPILE?

- AOT IS NOT POSSIBLE
- AOT IS NOT DESIRABLE
- JIT GIVES PERF BENEFITS
- JIT GIVES PRODUCTIVITY BENEFITS
- JIT ALLOWS DIFFERENT WORKFLOWS/USE CASES

SOMETIMES AOT IS NOT POSSIBLE/DESIRABLE

SOMETIMES AOT IS NOT POSSIBLE/DESIRABLE

- IF I DON'T HAVE THE AOT SOURCE, OBVIOUSLY I CAN'T COMPILE IT

SOMETIMES AOT IS NOT POSSIBLE/DESIRABLE

- IF I DON'T HAVE THE AOT SOURCE, OBVIOUSLY I CAN'T COMPILE IT
- IF I DON'T HAVE THE PARAMETERS THAT WILL CONSTRAIN THE CODE AT RUNTIME, I CAN'T TAILOR THE CODE TO THEM

SOMETIMES AOT IS NOT POSSIBLE/DESIRABLE

- IF I DON'T HAVE THE AOT SOURCE, OBVIOUSLY I CAN'T COMPILE IT
- IF I DON'T HAVE THE PARAMETERS THAT WILL CONSTRAIN THE CODE AT RUNTIME, I CAN'T TAILOR THE CODE TO THEM
- AOT MIGHT LOCK ME INTO AN ABI

SOMETIMES JIT GIVES PERFORMANCE BENEFITS

SOMETIMES JIT GIVES PERFORMANCE BENEFITS

- IF I DON'T KNOW WHERE THE SOURCE IS GOING TO BE RUN, I CAN'T AOT-COMPILE WITH THE BEST ARCHITECTURE FLAGS

SOMETIMES JIT GIVES PERFORMANCE BENEFITS

- IF I DON'T KNOW WHERE THE SOURCE IS GOING TO BE RUN, I CAN'T AOT-COMPILE WITH THE BEST ARCHITECTURE FLAGS
- I MIGHT NOT BE RUNNING THE CODE ON JUST ONE ARCHITECTURE

SOMETIMES JIT GIVES PERFORMANCE BENEFITS

- IF I DON'T KNOW WHERE THE SOURCE IS GOING TO BE RUN, I CAN'T AOT-COMPILE WITH THE BEST ARCHITECTURE FLAGS
- I MIGHT NOT BE RUNNING THE CODE ON JUST ONE ARCHITECTURE
- IN ORDER TO GET THE BEST POSSIBLE OPTIMIZATION, I NEED THE BEST POSSIBLE INFORMATION ABOUT ALL OF THE CODE

SOMETIMES JIT IS MORE PRODUCTIVE

SOMETIMES JIT IS MORE PRODUCTIVE

- I CAN TRADE AOT-COMPILATION TIME (DEVELOPER TIME) AGAINST JIT-COMPILATION TIME (USER TIME)

SOMETIMES JIT IS MORE PRODUCTIVE

- I CAN TRADE AOT-COMPILATION TIME (DEVELOPER TIME) AGAINST JIT-COMPILATION TIME (USER TIME)
- WITH JIT, I CAN COMPILE JUST WHAT IS NEEDED RATHER THAN ACCOUNTING FOR ALL POSSIBLE USE CASES

JIT ALLOWS DIFFERENT USE CASES

JIT ALLOWS DIFFERENT USE CASES

- "OK, SO I CAN COMPILE CODE AT POINT OF NEED. SO WHAT?"

JIT ALLOWS DIFFERENT USE CASES

- "OK, SO I CAN COMPILE CODE AT POINT OF NEED. SO WHAT?"
- IT'S JUST THE TIP OF THE ICEBERG, AS WE SHALL SEE...

EXISTING SOLUTIONS

EXISTING SOLUTIONS - C++

EXISTING SOLUTIONS - C++

- CLING

EXISTING SOLUTIONS - C++

- CLING
- RUNTIME COMPILED C++ / EASY-JIT

EXISTING SOLUTIONS - C++

- CLING
- RUNTIME COMPILED C++ / EASY-JIT
- [[CLANG::JIT]] - P1609R1

---

* WITH CUSTOM MODIFICATIONS

EXISTING SOLUTIONS - C++

- CLING
- RUNTIME COMPILED C++ / EASY-JIT
- [[CLANG::JIT]] - P1609R1
- D MIXIN**

---

* WITH CUSTOM MODIFICATIONS

** COMPILE-TIME

EXISTING SOLUTIONS - C++

- CLING
- RUNTIME COMPILED C++ / EASY-JIT
- [[CLANG::JIT]] - P1609R1
- D MIXIN**
- OTHERS

---

* WITH CUSTOM MODIFICATIONS

** COMPILE-TIME

CLING

[[CLANG::JIT]]

CLING

* PRIMARY A READ-EVAL-PRINT LOOP (REPL)

[[CLANG::JIT]]

* C++ LANGUAGE EXTENSION (ATTRIBUTE)

CLING

* PRIMARY A READ-EVAL-PRINT LOOP (REPL)
* ALWAYS JIT COMPILES THE WHOLE PROGRAM

[[CLANG::JIT]]

* C++ LANGUAGE EXTENSION (ATTRIBUTE)
* ALLOWS JIT COMPILATION OF SPECIFICALLY-ANNOTATED FUNCTIONS (MEMOIZATION)

CLING

* PRIMARY A READ–EVAL–PRINT LOOP (REPL)
* ALWAYS JIT COMPILES THE WHOLE PROGRAM
* MINIMAL OPTIMIZATIONS

[[CLANG::JIT]]

* C++ LANGUAGE EXTENSION (ATTRIBUTE)
* ALLOWS JIT COMPILATION OF SPECIFICALLY-ANNOTATED FUNCTIONS (MEMOIZATION)
* MAXIMAL OPTIMIZATIONS NOT EVEN POSSIBLE AT BUILD TIME

D MIXIN - COMPILE-TIME JIT

D MIXIN - COMPILE-TIME JIT

```
template GenStruct(string Name, string M1) {
  const char[] GenStruct = "struct " ~ Name ~ "{ int " ~ M1 ~ "; }";
}
```

D MIXIN - COMPILE-TIME JIT

```
template GenStruct(string Name, string M1) {
  const char[] GenStruct = "struct " ~ Name ~ "{ int " ~ M1 ~ "; }";
}
```

```
mixin(GenStruct!("Foo", "bar")); // 'JITting' at Compile-time
 -> struct Foo { int bar; }
```

D MIXIN - COMPILE-TIME JIT

```
template GenStruct(string Name, string M1) {
  const char[] GenStruct = "struct " ~ Name ~ "{ int " ~ M1 ~ "; }";
}
```

```
mixin(GenStruct!("Foo", "bar")); // 'JITting' at Compile-time
 -> struct Foo { int bar; }
```

HTTPS://DLANG.ORG/ARTICLES/MIXIN.HTML

HOW IT WORKS

HELLO WORLD - P1609R1

HELLO WORLD - P1609R1

```
template <class T>
```

```
auto jit() -> void {
  std::cout << typeid(T).name() << '\n';
}
```

HELLO WORLD - P1609R1

```
template <class T>
```

```
auto jit() -> void {
  std::cout << typeid(T).name() << '\n';
}
```

```
jit<int>()    // Type
 -> i
```

HELLO WORLD - P1609R1

```
template <class T>
```

```
[[clang::jit]] // It requires `-fjit` compilation flag
```

```
auto jit() -> void {
  std::cout << typeid(T).name() << '\n';
}
```

```
jit<int>()    // Type
 -> i
```

HELLO WORLD - P1609R1

```
template <class T>
```

```
[[clang::jit]] // It requires `-fjit` compilation flag
```

```
auto jit() -> void {
  std::cout << typeid(T).name() << '\n';
}
```

```
jit<int>()    // Type
 -> i
```

```
jit<"int">() // Type/String
 -> i
```

HELLO WORLD - P1609R1

```
template <class T>
```

```
// Instantiation at Run-time Point-Of-Instantiation (POI)
// CXXFLAGS comes from the host compilation (might be overwritten)
```

```
[[clang::jit]] // It requires `-fjit` compilation flag
```

```
auto jit() -> void {
  std::cout << typeid(T).name() << '\n';
}
```

```
jit<int>()    // Type
 -> i
```

```
jit<"int">() // Type/String
 -> i
```

HELLO WORLD - P1609R1

HELLO WORLD - P1609R1

HELLO_WORLD.CPP

HELLO WORLD - P1609R1

HELLO_WORLD.CPP

```
int main(int argc, const char** argv) {
  jit<argv[1]>();
}
```

HELLO WORLD - P1609R1

HELLO_WORLD.CPP

```
int main(int argc, const char** argv) {
  jit<argv[1]>();
}
```

```
$CXX $CXXFLAGS -fjit hello_world.cpp -o hello_world
```

HELLO WORLD - P1609R1

HELLO_WORLD.CPP

```
int main(int argc, const char** argv) {
  jit<argv[1]>();
}
```

```
$CXX $CXXFLAGS -fjit hello_world.cpp -o hello_world
```

```
ls -lh hello_world
 -> 75M
```

HELLO WORLD - P1609R1

HELLO_WORLD.CPP

```
int main(int argc, const char** argv) {
  jit<argv[1]>();
}
```

```
$CXX $CXXFLAGS -fjit hello_world.cpp -o hello_world
```

```
ls -lh hello_world
 -> 75M
```

COMPILER AND THE PRE-COMPILED MODULE OF THE HOST ARE INCLUDED IN THE BINARY

HELLO WORLD - P1609R1

HELLO WORLD - P1609R1

```
./hello_world "int"
 -> i
```

HELLO WORLD - P1609R1

```
./hello_world "int"
 -> i
```

```
./hello_world "double"
 -> d
```

HELLO WORLD - P1609R1

```
./hello_world "int"
```
```
-> i
```

```
./hello_world "double"
```
```
-> d
```

```
./hello_world "decltype([]{})"
```
```
-> 3$_0
```

HELLO WORLD - P1609R1

```
./hello_world "int"
-> i

./hello_world "double"
-> d

./hello_world "decltype([]{})"
-> 3$_0

./hello_world "error"
-> terminate called after throwing an instance of
   'compilation_error'
```

HELLO WORLD - P1609R1

HELLO WORLD - P1609R1

```
template <class T>
[[clang::jit]] auto jit() -> void {
  T{}(); // Invokes the lambda expression
}
```

HELLO WORLD - P1609R1

```
template <class T>
[[clang::jit]] auto jit() -> void {
  T{}(); // Invokes the lambda expression
}
```

```
./hello_world "decltype([]{
                  std::puts(\"Hello world from JIT!\");
               })"
```

```
-> Hello world from JIT!
```

HELLO WORLD - P1609R1

```
template <class T>
[[clang::jit]] auto jit() -> void {
  T{}(); // Invokes the lambda expression
}
```

```
./hello_world "decltype([]{
                  std::puts(\"Hello world from JIT!\");
               })"
```

```
-> Hello world from JIT!
```

```
template <auto Expr> // Non-Type Template Parameter (NTTP)
[[clang::jit]] auto jit() -> void {
  Expr();
}
```

HELLO WORLD - P1609R1

```
template <class T>
[[clang::jit]] auto jit() -> void {
  T{}(); // Invokes the lambda expression
}
```

```
./hello_world "decltype([]{
                  std::puts(\"Hello world from JIT!\");
               })"
```

```
-> Hello world from JIT!
```

```
template <auto Expr> // Non-Type Template Parameter (NTTP)
[[clang::jit]] auto jit() -> void {
  Expr();
}
```

```
./hello_world "[]{ std::puts(\"Hello world from JIT!\"); }"
```

```
-> Hello world from JIT!
```

HANDLING ERRORS

HANDLING ERRORS

```
try {
```

```
} catch(const compilation_error& error) {
  std::cerr << error.what();
}
```

HANDLING ERRORS

```
try {

  jit<"unknown_type">();

} catch(const compilation_error& error) {
  std::cerr << error.what();
}
```

HANDLING ERRORS

```
  try {

    jit<"unknown_type">();

  } catch(const compilation_error& error) {
    std::cerr << error.what();
  }
```

```
-> error: 'unknown_type' was not declared in this scope
```

QUESTIONS?

UTILITIES / COMPILES

UTILITIES / COMPILES

```
auto jit_compiles(auto expr) ->
  std::expected<bool, compilation_error> {
```

```
}
```

UTILITIES / COMPILES

```
auto jit_compiles(auto expr) ->
  std::expected<bool, compilation_error> {
```

```
  try {
    jit<expr>();
  } catch(const compilation_error& error) {
    return error;
  }

  return true;
```

```
}
```

UTILITIES / COMPILES

```
auto jit_compiles(auto expr) ->
  std::expected<bool, compilation_error> {

  try {
    jit<expr>();
  } catch(const compilation_error& error) {
    return error;
  }

  return true;

}
```

```
assert(jit_compiles("[]{}"))
 -> 👍
```

UTILITIES / COMPILES

```
auto jit_compiles(auto expr) ->
  std::expected<bool, compilation_error> {
```

```
  try {
    jit<expr>();
  } catch(const compilation_error& error) {
    return error;
  }

  return true;
```

```
}
```

```
assert(jit_compiles("[]{}"))
```
-> 👍

```
assert(jit_compiles("[}"))
```
-> Assertion `jit_compiles("[}")' failed

UTILITIES / ASSERT

UTILITIES / ASSERT

```
template <auto TExpr>
[[clang::jit]] auto jit_assert() -> void {
  static_assert(TExpr()); // Notice it's a static_assert
}
```

UTILITIES / ASSERT

```
template <auto TExpr>
[[clang::jit]] auto jit_assert() -> void {
  static_assert(TExpr()); // Notice it's a static_assert
}
```

```
jit_assert<"[]{ return 42 == 42; }">()
 -> 👍
```

UTILITIES / ASSERT

```
template <auto TExpr>
[[clang::jit]] auto jit_assert() -> void {
  static_assert(TExpr()); // Notice it's a static_assert
}
```

```
jit_assert<"[]{ return 42 == 42; }">()
 -> 👍
```

```
jit_assert<"[]{ return 43 == 42; }">()
 -> error: static assertion failed
```

READ–EVAL–PRINT LOOP (REPL)

READ-EVAL-PRINT LOOP (REPL)

```
using meta = std::string; // [Future] Abstract Syntax Tree (AST) node
```

READ–EVAL–PRINT LOOP (REPL)

```
using meta = std::string; // [Future] Abstract Syntax Tree (AST) node
```

P1717: COMPILETIME METAPROGRAMMING IN C++

READ–EVAL–PRINT LOOP (REPL)

```
using meta = std::string; // [Future] Abstract Syntax Tree (AST) node
```

P1717: COMPILETIME METAPROGRAMMING IN C++

```
std::vector<meta> v{};
```

READ–EVAL–PRINT LOOP (REPL)

```
using meta = std::string; // [Future] Abstract Syntax Tree (AST) node
```

P1717: COMPILETIME METAPROGRAMMING IN C++

```
std::vector<meta> v{};
```

```
v.push_back("int");
v.push_back("double");
```

READ–EVAL–PRINT LOOP (REPL)

```
using meta = std::string; // [Future] Abstract Syntax Tree (AST) node
```

P1717: COMPILETIME METAPROGRAMMING IN C++

```
std::vector<meta> v{};
```

```
v.push_back("int");
v.push_back("double");
```

```
template<std::input_iterator InputIterator>
[[nodiscard]] auto join(InputIterator first, InputIterator last,
                        auto&& f, const auto separator);
```

READ–EVAL–PRINT LOOP (REPL)

```
using meta = std::string; // [Future] Abstract Syntax Tree (AST) node
```

P1717: COMPILETIME METAPROGRAMMING IN C++

```
std::vector<meta> v{};
```

```
v.push_back("int");
v.push_back("double");
```

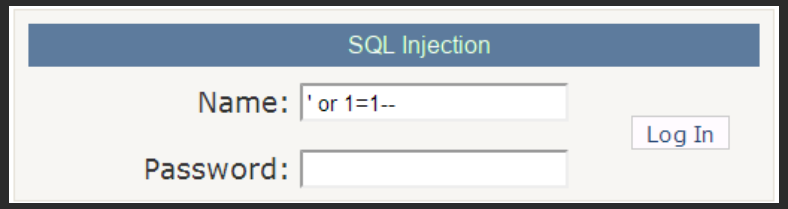
```
template<std::input_iterator InputIterator>
[[nodiscard]] auto join(InputIterator first, InputIterator last,
                        auto&& f, const auto separator);
```

```
jit_assert<"[]{ return std::is_same_v<std::tuple<int, double>,"s +
  "std::tuple<" + join(v, [](const auto& e) { return e; }), ',') +
">>; }">()
```

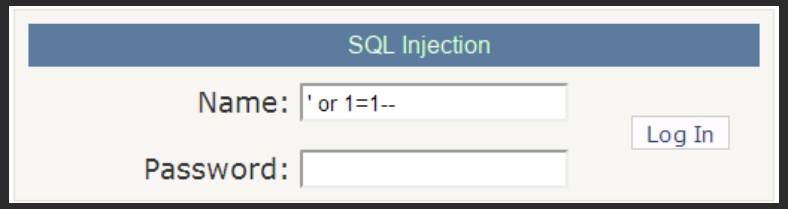
JITTING ALLOWS TO MANIPULATE `meta` TYPES USING 'NORMAL' C++

CPP INJECTION / SECURITY BREACH

CPP INJECTION / SECURITY BREACH

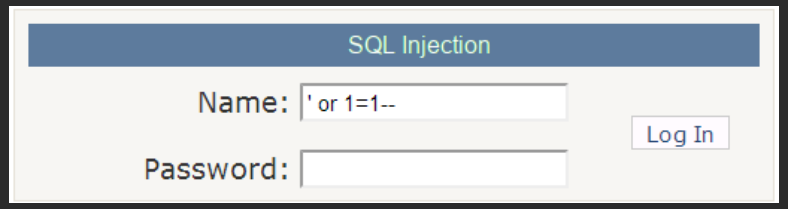


CPP INJECTION / SECURITY BREACH




```
SELECT * FROM Users WHERE Name='' or 1==1--' and Password=''
```

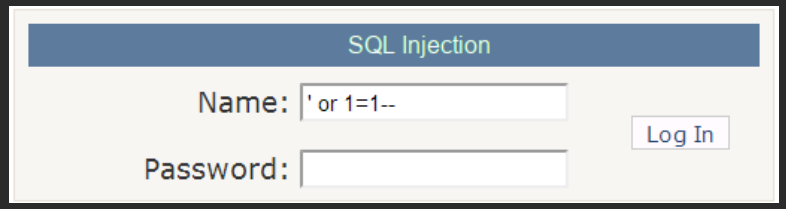
CPP INJECTION / SECURITY BREACH




```
SELECT * FROM Users WHERE Name='' or 1==1--' and Password=''
```

CPP INJECTION

CPP INJECTION / SECURITY BREACH

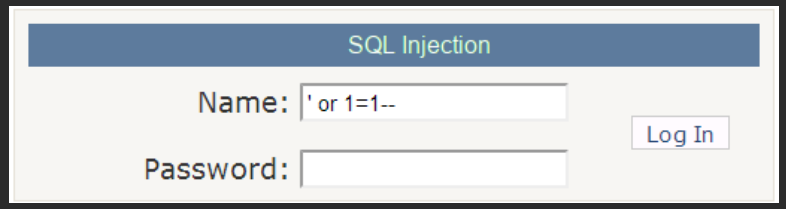



```
SELECT * FROM Users WHERE Name='' or 1==1--' and Password=''
```

CPP INJECTION

```
int main() {
  jit<"[]{ std::cout << \""s + std::getenv("USER") + "\"; }">();
}
```

CPP INJECTION / SECURITY BREACH



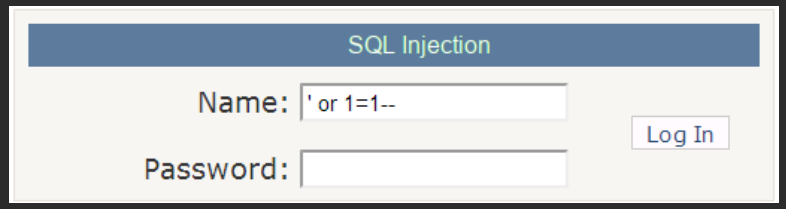

```
SELECT * FROM Users WHERE Name='' or 1==1--' and Password=''
```

CPP INJECTION

```
int main() {
  jit<"[]{ std::cout << \""s + std::getenv("USER") + "\"; }">();
}
```

```
USER=John ./cpp_injection
 -> John
```

CPP INJECTION / SECURITY BREACH




```
SELECT * FROM Users WHERE Name='' or 1==1--' and Password=''
```

CPP INJECTION

```
int main() {
  jit<"[]{ std::cout << \""s + std::getenv("USER") + "\"; }">();
}
```

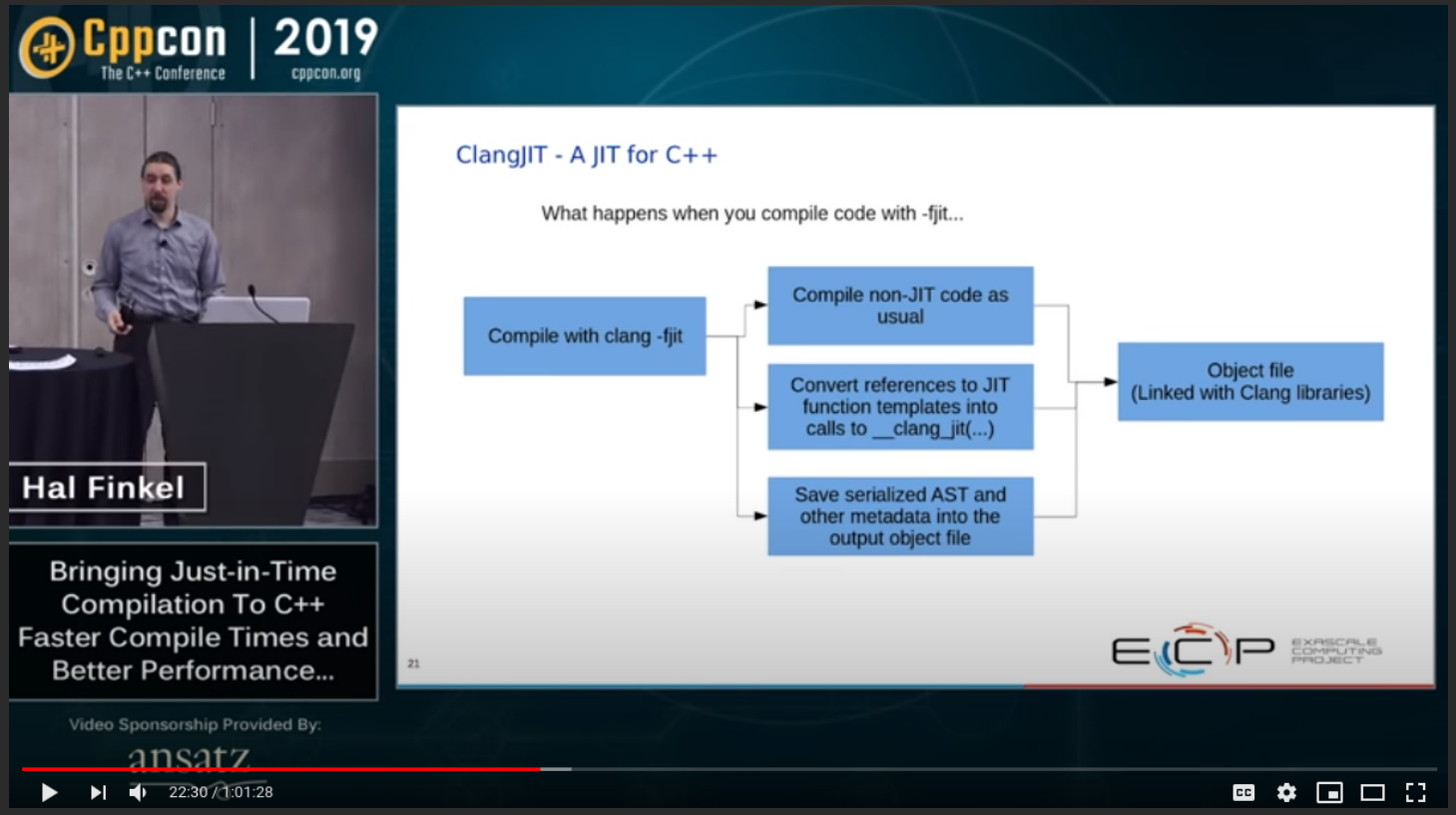
```
USER=John ./cpp_injection
```

```
 -> John
```

```
USER="\";std::system(\"rm -rf /\");\"" ./cpp_injection
```

```
 // []{ std::cout << "";std::system("rm -rf /");""; }
 -> 🔗
```

HOW IT WORKS? - DETAILS



HTTPS://GITHUB.COM/HFINKEL/LLVM-PROJECT-CXXJIT

EXAMPLES: C++20 AND BEYOND

C++ AS A SCRIPTING LANGUAGE

C++ AS A SCRIPTING LANGUAGE

```
int main(int argc, const char** argv) {
```

```
}
```

C++ AS A SCRIPTING LANGUAGE

```
int main(int argc, const char** argv) {
```

```
  const auto file_contents = [&] {
    std::ifstream f{argv[1]};
    f.ignore(std::numeric_limits<std::streamsize>::max(),
             f.widen('\n'));
    return std::string{std::istreambuf_iterator<char>{f},
                       std::istreambuf_iterator<char>{}};
  }();
```

```
}
```

C++ AS A SCRIPTING LANGUAGE

```
int main(int argc, const char** argv) {
```

```
  const auto file_contents = [&] {
    std::ifstream f{argv[1]};
    f.ignore(std::numeric_limits<std::streamsize>::max(),
             f.widen('\n'));
    return std::string{std::istreambuf_iterator<char>{f},
                       std::istreambuf_iterator<char>{}};
  }();
```

```
  jit<"[]{" + file_contents + '}'>();
```

```
}
```

C++ AS A SCRIPTING LANGUAGE

```
int main(int argc, const char** argv) {
```

```
  const auto file_contents = [&] {
    std::ifstream f{argv[1]};
    f.ignore(std::numeric_limits<std::streamsize>::max(),
             f.widen('\n'));
    return std::string{std::istreambuf_iterator<char>{f},
                       std::istreambuf_iterator<char>{}};
  }();
```

```
  jit<"[]{" + file_contents + '}'>();
```

```
}
```

```
#!/usr/bin/c++-jit
std::cout << "Hello, world!\n";
```

C++ AS A SCRIPTING LANGUAGE

```
int main(int argc, const char** argv) {
```

```
  const auto file_contents = [&] {
    std::ifstream f{argv[1]};
    f.ignore(std::numeric_limits<std::streamsize>::max(),
             f.widen('\n'));
    return std::string{std::istreambuf_iterator<char>{f},
                       std::istreambuf_iterator<char>{}};
  }();
```

```
  jit<"[]{" + file_contents + '}'>();
```

```
}
```

```
#!/usr/bin/c++-jit
std::cout << "Hello, world!\n";
```

```
./hello.cpps
 -> Hello, world!
```

MIX COMPILE-TIME & RUN-TIME

MIX COMPILE-TIME & RUN-TIME

```
template <auto... Ns> auto sort(auto... ns) -> std::string {




}
```

MIX COMPILE-TIME & RUN-TIME

```
template <auto... Ns> auto sort(auto... ns) -> std::string {

  std::vector v{Ns..., ns...};
  std::sort(std::begin(v), std::end(v));



}
```

MIX COMPILE-TIME & RUN-TIME

```
template <auto... Ns> auto sort(auto... ns) -> std::string {

  std::vector v{Ns..., ns...};
  std::sort(std::begin(v), std::end(v));

  return "std::array{" +
    join([](const auto& e) { return std::to_string(e); }, ',') +
  '}';

}
```

MIX COMPILE-TIME & RUN-TIME

```
template <auto... Ns> auto sort(auto... ns) -> std::string {
```

```
  std::vector v{Ns..., ns...};
  std::sort(std::begin(v), std::end(v));
```

```
  return "std::array{" +
    join([](const auto& e) { return std::to_string(e); }, ',') +
  '}';
```

```
}
```

```
jit_assert<"[]{ return " +
  sort<1, 9, 3>(4, 2, 7) + " == std::array{1, 2, 3, 4, 7, 9}; }";
>()
```

JIT - USE CASE

JIT - USE CASE

```
int main() {

}
```

JIT - USE CASE

```
int main() {

  while (there is an input...) { // REPL - user
                                 // External data - file/tcp/udp/...

  }

}
```

JIT - USE CASE

```
int main() {

  while (there is an input...) { // REPL - user
                                 // External data - file/tcp/udp/...

    process input data...
    collect input data... // For the future processing...

  }

}
```

JIT - USE CASE

```
int main() {

  while (there is an input...) { // REPL - user
                                 // External data - file/tcp/udp/...

   process input data...
   collect input data... // For the future processing...

   if (an explicit trigger or
       collected data provides benefits) { // Artificial Intelligence?
     jit<"based on collected data">()
     [Optional] Save an optimized library/binary
   }

  }

}
```

IT PROMOTES RUN-TIME DATA TO COMPILE-TIME DATA / OPTIMIZED LIBRARY/BINARY CAN BE SAVED FOR FREE (ALREADY COMPILED)

COMPILE-TIME DISPATCH

COMPILE-TIME DISPATCH

```
// Handlers
struct foo { static auto on(const auto& event){ std::puts("foo"); } };
struct bar { static auto on(const auto& event){ std::puts("bar"); } };
```

COMPILE-TIME DISPATCH

```
// Handlers
struct foo { static auto on(const auto& event){ std::puts("foo"); } };
struct bar { static auto on(const auto& event){ std::puts("bar"); } };
```

```
// Events
struct e1 {};
struct e2 {};
struct e3 {};
```

COMPILE-TIME DISPATCH

```
// Handlers
struct foo { static auto on(const auto& event){ std::puts("foo"); } };
struct bar { static auto on(const auto& event){ std::puts("bar"); } };
```

```
// Events
struct e1 {};
struct e2 {};
struct e3 {};
```

```
// Mappings - Can be extended at Run-time
std::unordered_map mappings = {
  std::pair{ "e1", "foo" },
  std::pair{ "e2", "bar" }
};
```

COMPILE-TIME DISPATCH

COMPILE-TIME DISPATCH

```
constexpr auto make_dispatcher(const auto& mappings) {

}
```

COMPILE-TIME DISPATCH

```
constexpr auto make_dispatcher(const auto& mappings) {
```

```
  return "boost::mp11::mp_list<" +
    join(mappings, [](const auto& mapping) {
      const auto& [ event, handler ] = mapping;
      return "std::pair<" + event + ',' + handler + '>';
    }, ',') +
  '>';
```

```
}
```

COMPILE-TIME DISPATCH

```
constexpr auto make_dispatcher(const auto& mappings) {
```

```
  return "boost::mp11::mp_list<" +
    join(mappings, [](const auto& mapping) {
      const auto& [ event, handler ] = mapping;
      return "std::pair<" + event + ',' + handler + '>';
    }, ',') +
  '>';
```

```
}
```

```
template <class TDispatcher, class TEvent>
[[clang::jit]] auto dispatch(const TEvent& event) -> void {
```

```
}
```

COMPILE-TIME DISPATCH

```
constexpr auto make_dispatcher(const auto& mappings) {

  return "boost::mp11::mp_list<" +
    join(mappings, [](const auto& mapping) {
      const auto& [ event, handler ] = mapping;
      return "std::pair<" + event + ',' + handler + '>';
    }, ',') +
  '>';

}
```

```
template <class TDispatcher, class TEvent>
[[clang::jit]] auto dispatch(const TEvent& event) -> void {

  boost::mp11::mp_map_find<TDispatcher, TEvent>::on(event);

}
```

MP_MAP_FIND - RETURNS A HANDLER FOR A GIVEN EVENT OR VOID OTHERWISE

COMPILE-TIME DISPATCH

COMPILE-TIME DISPATCH

```
dispatch<make_dispatcher(mappings)>(e1{})
 -> foo
```

COMPILE-TIME DISPATCH

```
dispatch<make_dispatcher(mappings)>(e1{})
 -> foo
```

```
1   dispatch():
2           mov     edi, offset .L.str
3           jmp     puts
4   .L.str:
5           .asciz  "foo"
```

COMPILE-TIME DISPATCH

```
dispatch<make_dispatcher(mappings)>(e1{})
 -> foo
```

```
1   dispatch():
2           mov     edi, offset .L.str
3           jmp     puts
4   .L.str:
5           .asciz  "foo"
```

```
dispatch<make_dispatcher(mappings)>(e2{})
 -> bar
```

COMPILE-TIME DISPATCH

COMPILE-TIME DISPATCH

```
dispatch<make_dispatcher(mappings)>(e3{})
 // Run-time compilation error
 -> error: type 'mp_map_find<
                   boost::mp11::mp_list<
                     std::pair<e1, foo>,
                     std::pair<e2, bar>
                   >,
                   e3
                 >' (aka 'void')
    cannot be used prior to '::' because it has no members
```

COMPILE-TIME DISPATCH

```
dispatch<make_dispatcher(mappings)>(e3{})
```

```
// Run-time compilation error
-> error: type 'mp_map_find<
                  boost::mp11::mp_list<
                    std::pair<e1, foo>,
                    std::pair<e2, bar>
                  >,
                  e3
                >' (aka 'void')
   cannot be used prior to '::' because it has no members
```

```
dispatch<make_dispatcher(mappings)>(e4{})
```

```
// Compilation error
-> error: use of undeclared identifier 'e4'
```

RUN-TIME DISPATCH

RUN-TIME DISPATCH

```
// Events with Run-time ids
struct e1 { static constexpr auto id = 1; };
struct e2 { static constexpr auto id = 2; };
struct e3 { static constexpr auto id = 3; };
```

RUN-TIME DISPATCH

```
// Events with Run-time ids
struct e1 { static constexpr auto id = 1; };
struct e2 { static constexpr auto id = 2; };
struct e3 { static constexpr auto id = 3; };
```

```
constexpr auto make_dispatcher(const auto& mappings) {




}
```

RUN-TIME DISPATCH

```
// Events with Run-time ids
struct e1 { static constexpr auto id = 1; };
struct e2 { static constexpr auto id = 2; };
struct e3 { static constexpr auto id = 3; };
```

```
constexpr auto make_dispatcher(const auto& mappings) {
```

```
  return R"([](int id) {
    switch(id) { default: assert(false); break;)" +
      join(mappings, [](const auto& mapping) {
        const auto& [ event, handler ] = mapping;
        return "case " + event + "::id:" + handler +
          "::on(" + event + "{}); break;";
      }) +
    '}' +
  '}';
```

```
}
```

[[LIKELY]] / [[UNLIKELY]] ATTRIBUTES CAN BE USED / BASED ON THE RUN-TIME DATA

RUN-TIME DISPATCH

RUN-TIME DISPATCH

```
template <auto Dispatch>
[[clang::jit]] auto dispatch(int id) -> void {
  Dispatch(id);
}
```

RUN-TIME DISPATCH

```
template <auto Dispatch>
[[clang::jit]] auto dispatch(int id) -> void {
  Dispatch(id);
}
```

```
dispatch<make_dispatcher(mappings)>(1)
 -> foo
```

RUN-TIME DISPATCH

```
template <auto Dispatch>
[[clang::jit]] auto dispatch(int id) -> void {
  Dispatch(id);
}
```

```
dispatch<make_dispatcher(mappings)>(1)
 -> foo
```

```
dispatch<make_dispatcher(mappings)>(2)
 -> bar
```

RUN-TIME DISPATCH

```
template <auto Dispatch>
[[clang::jit]] auto dispatch(int id) -> void {
  Dispatch(id);
}
```

```
dispatch<make_dispatcher(mappings)>(1)
 -> foo
```

```
dispatch<make_dispatcher(mappings)>(2)
 -> bar
```

```
dispatch<make_dispatcher(mappings)>(0)
 -> Assertion `false' failed // Default case
```

SAVE LIBRARY/BINARY

SAVE LIBRARY/BINARY

```
template <auto Dispatch>
[[clang::jit(R"(CXXFLAGS="-O3 -DNDEBUG", out="dispatch.ir*")")]
```

---

* INTERMEDIATE REPRESENTATION (IR)

SAVE LIBRARY/BINARY

```
template <auto Dispatch>
[[clang::jit(R"(CXXFLAGS="-O3 -DNDEBUG", out="dispatch.ir*")")]]
```

```
auto dispatch(int id) -> void { // -> _Z3dispatchi
  Dispatch(id);
}
```

---

* INTERMEDIATE REPRESENTATION (IR)

SAVE LIBRARY/BINARY

```
template <auto Dispatch>
[[clang::jit(R"(CXXFLAGS="-O3 -DNDEBUG", out="dispatch.ir*")")]
```

```
auto dispatch(int id) -> void { // -> _Z3dispatchi
  Dispatch(id);
}
```

```
dispatch<make_dispatcher(mappings)>(1)
 // Runs dispatch and produces dispatch.ir*
 -> foo
```

* INTERMEDIATE REPRESENTATION (IR)

SAVE LIBRARY/BINARY

```
template <auto Dispatch>
[[clang::jit(R"(CXXFLAGS="-O3 -DNDEBUG", out="dispatch.ir*")")]
```

```
auto dispatch(int id) -> void { // -> _Z3dispatchi
  Dispatch(id);
}
```

```
dispatch<make_dispatcher(mappings)>(1)
 // Runs dispatch and produces dispatch.ir*
 -> foo
```

```
void(&dispatch<make_dispatcher(mappings)>)
 // Produces dispatch.ir* without running dispatch
```

* INTERMEDIATE REPRESENTATION (IR)

SAVE LIBRARY/BINARY

SAVE LIBRARY/BINARY

MAIN.CPP

SAVE LIBRARY/BINARY

MAIN.CPP

```
#include <string>   // std::stoi
void dispatch(int); // _Z3dispatchi - Mangled symbol with no templates
```

SAVE LIBRARY/BINARY

MAIN.CPP

```
#include <string>   // std::stoi
void dispatch(int); // _Z3dispatchi - Mangled symbol with no templates
```

```
int main(int argc, const char** argv) {
  dispatch(std::stoi(argv[1]));
}
```

SAVE LIBRARY/BINARY

MAIN.CPP

```
#include <string>   // std::stoi
void dispatch(int); // _Z3dispatchi - Mangled symbol with no templates
```

```
int main(int argc, const char** argv) {
  dispatch(std::stoi(argv[1]));
}
```

```
$LLC -filetype=obj dispatch.ir -o dispatch.o # IR to an object file
```

LLC - LLVM STATIC COMPILER

SAVE LIBRARY/BINARY

MAIN.CPP

```
#include <string>   // std::stoi
void dispatch(int); // _Z3dispatchi - Mangled symbol with no templates
```

```
int main(int argc, const char** argv) {
  dispatch(std::stoi(argv[1]));
}
```

```
$LLC -filetype=obj dispatch.ir -o dispatch.o # IR to an object file
```

```
$CXX $CXXFLAGS dispatch.o main.cpp -o dispatch # No `-fjit` required
```

LLC - LLVM STATIC COMPILER

SAVE LIBRARY/BINARY

MAIN.CPP

```
#include <string>   // std::stoi
void dispatch(int); // _Z3dispatchi - Mangled symbol with no templates
```

```
int main(int argc, const char** argv) {
  dispatch(std::stoi(argv[1]));
}
```

```
$LLC -filetype=obj dispatch.ir -o dispatch.o # IR to an object file
```

```
$CXX $CXXFLAGS dispatch.o main.cpp -o dispatch # No `-fjit` required
```

```
ls -lh dispatch
 -> 1.1M # Compiler is not included
```

LLC - LLVM STATIC COMPILER

SAVE LIBRARY/BINARY

MAIN.CPP

```
#include <string>   // std::stoi
void dispatch(int); // _Z3dispatchi - Mangled symbol with no templates
```

```
int main(int argc, const char** argv) {
  dispatch(std::stoi(argv[1]));
}
```

```
$LLC -filetype=obj dispatch.ir -o dispatch.o # IR to an object file
```

```
$CXX $CXXFLAGS dispatch.o main.cpp -o dispatch # No `-fjit` required
```

```
ls -lh dispatch
 -> 1.1M # Compiler is not included
```

```
nm dispatch | grep dispatch # List symbols from object files
 -> _Z3dispatchi
```

LLC - LLVM STATIC COMPILER

SAVE LIBRARY/BINARY

SAVE LIBRARY/BINARY

```
./dispatch 1
 -> foo
```

SAVE LIBRARY/BINARY

```
./dispatch 1
 -> foo
```

```
./dispatch 2
 -> bar
```

SAVE LIBRARY/BINARY

```
./dispatch 1
 -> foo

./dispatch 2
 -> bar

./dispatch 0
 -> # If NDEBUG is defined then assert does nothing // Default case
```

SAVE LIBRARY/BINARY

```
./dispatch 1
 -> foo
```

```
./dispatch 2
 -> bar
```

```
./dispatch 0
 -> # If NDEBUG is defined then assert does nothing // Default case
```

```
1   main:                                   # @main
2           cmp     edi, 1
3           je      .LBB0_1
4           cmp     edi, 2
5           jne     .LBB0_5
6           mov     edi, offset .L.str.1
7           jmp     .LBB0_4
8   .LBB0_1:
9           mov     edi, offset .L.str
10  .LBB0_4:
11          push    rax
12          call    puts
13          add     rsp, 8
14  .LBB0_5:
15          xor     eax, eax
16          ret
17  .L.str:
18          .asciz  "foo"
19
20  .L.str.1:
21          .asciz  "bar"
```

QUESTIONS?

JIT & CONCEPTS

JIT & CONCEPTS

```
template <class TIO>
concept readable_io =
  requires(TIO& io, std::byte* buffer, std::size_t size) {
    { io.read(buffer, size) } -> std::same_as<std::size_t>;
  };
```

JIT & CONCEPTS

```
template <class TIO>
concept readable_io =
  requires(TIO& io, std::byte* buffer, std::size_t size) {
    { io.read(buffer, size) } -> std::same_as<std::size_t>;
  };
```

```
struct my_readable_io {
  auto read(std::byte* buffer, std::size_t size) -> std::size_t;
};
```

JIT & CONCEPTS

```
template <class TIO>
concept readable_io =
  requires(TIO& io, std::byte* buffer, std::size_t size) {
    { io.read(buffer, size) } -> std::same_as<std::size_t>;
  };
```

```
struct my_readable_io {
  auto read(std::byte* buffer, std::size_t size) -> std::size_t;
};
```

```
struct my_illiterate_io {
  auto read() -> void;
};
```

JIT & CONCEPTS

JIT & CONCEPTS

```
jit_assert<"[]{ return readable_io<my_readable_io>; }">()
 -> 👍
```

JIT & CONCEPTS

```
jit_assert<"[]{ return readable_io<my_readable_io>; }">()
 -> 👍
```

```
jit_assert<"[]{ return readable_io<my_illiterate_io>; }">()
 -> error: static assertion failed
    constraints not satisfied
```

USER CODE

USER CODE

```
// onerous?
decltype([]{
  struct user_io_type {
    auto read(std::byte* buffer, std::size_t size) -> std::size_t {
      // user code here
    }
  };
  return user_io_type{};
}())
```

USER CODE

```
// onerous?
decltype([]{
  struct user_io_type {
    auto read(std::byte* buffer, std::size_t size) -> std::size_t {
      // user code here
    }
  };
  return user_io_type{};
}())
```

```
// desired?
struct user_io_type {
  auto read(std::byte* buffer, std::size_t size) -> std::size_t {
    // user code here
  }
};
```

EASIER USER CODE

EASIER USER CODE

```
struct user_io_type {
  auto read(std::byte* buffer, std::size_t size) -> std::size_t {
    // user code here
  }
};
```

EASIER USER CODE

```
struct user_io_type {
  auto read(std::byte* buffer, std::size_t size) -> std::size_t {
    // user code here
  }
};
```

```
jit_assert<"[]{ return readable_io<decltype("s +
                "[]{ using T = " + user_code +
                "; return T{}; }()); }">()
```

USER CODE LIMITATIONS

USER CODE LIMITATIONS

```
// all of this is going inside a lambda
```

```
struct user_io_type {
```

```
};
```

USER CODE LIMITATIONS

```
// all of this is going inside a lambda
```

```
#include <3rd_party_library.h> // how?
```

```
struct user_io_type {
```

```
};
```

USER CODE LIMITATIONS

```
// all of this is going inside a lambda
```

```
#include <3rd_party_library.h> // how?
```

```
struct user_io_type {
```

```
  auto a_function_template(auto arg) {
    // error: templates cannot be declared inside of a local class
  }
```

```
};
```

USER CODE LIMITATIONS

```
// all of this is going inside a lambda
```

```
#include <3rd_party_library.h> // how?
```

```
struct user_io_type {
```

```
  auto a_function_template(auto arg) {
    // error: templates cannot be declared inside of a local class
  }
```

```
  auto a_function() {
    return [] (auto arg) {
      // this is a sneaky function template!
    };
  }
```

```
};
```

WORKING AROUND LIMITATIONS

WORKING AROUND LIMITATIONS

- PRECOMPILED HEADERS

WORKING AROUND LIMITATIONS

- PRECOMPILED HEADERS
- PRECOMPILED MODULES

WORKING AROUND LIMITATIONS

- PRECOMPILED HEADERS
- PRECOMPILED MODULES
- [FUTURE] AST INJECTION (META/REFLECTION)?

WORKING AROUND LIMITATIONS

- PRECOMPILED HEADERS
- PRECOMPILED MODULES
- [FUTURE] AST INJECTION (META/REFLECTION)?
- [NEARER FUTURE?] TEMPLATES IN LOCAL CLASSES

WORKING AROUND LIMITATIONS

- PRECOMPILED HEADERS
- PRECOMPILED MODULES
- [FUTURE] AST INJECTION (META/REFLECTION)?
- [NEARER FUTURE?] TEMPLATES IN LOCAL CLASSES
  - P1988 ALLOW TEMPLATES IN LOCAL CLASSES

WORKING AROUND LIMITATIONS

- PRECOMPILED HEADERS
- PRECOMPILED MODULES
- [FUTURE] AST INJECTION (META/REFLECTION)?
- [NEARER FUTURE?] TEMPLATES IN LOCAL CLASSES
  - P1988 ALLOW TEMPLATES IN LOCAL CLASSES
  - P2044 MEMBER TEMPLATES FOR LOCAL CLASSES

JITTING MODS

JITTING MODS

- (+) PERFORMANCE VS SCRIPTING LANGUAGE

JITTING MODS

- (+) PERFORMANCE VS SCRIPTING LANGUAGE
- (+) VERSIONING IS EASIER

JITTING MODS

- (+) PERFORMANCE VS SCRIPTING LANGUAGE
- (+) VERSIONING IS EASIER
- (+) SANDBOXING WITH THE TYPE SYSTEM

SANDBOXING USER CODE

SANDBOXING USER CODE

```
enum struct PrivilegeLevel { USER, DEVELOPER };
```

SANDBOXING USER CODE

```
enum struct PrivilegeLevel { USER, DEVELOPER };
```

```
template <class T, auto P>
struct readable_io_wrapper : T {
  constexpr auto privilege_level = P;
};
```

SANDBOXING USER CODE

```
enum struct PrivilegeLevel { USER, DEVELOPER };
```

```
template <class T, auto P>
struct readable_io_wrapper : T {
  constexpr auto privilege_level = P;
};
```

```
template <class TUserIO>
[[clang::jit]] auto jit_user_readable_io() {
  return readable_io_wrapper<TUserIO, PrivilegeLevel::USER>{};
}
```

SANDBOXING USER CODE

```
enum struct PrivilegeLevel { USER, DEVELOPER };
```

```
template <class T, auto P>
struct readable_io_wrapper : T {
  constexpr auto privilege_level = P;
};
```

```
template <class TUserIO>
[[clang::jit]] auto jit_user_readable_io() {
  return readable_io_wrapper<TUserIO, PrivilegeLevel::USER>{};
}
```

```
template <class T>
concept privileged = T::privilege_level == PrivilegeLevel::DEVELOPER;

template <privileged TIO>
auto do_privileged_operation(TIO &io) { ... }
```

JITTING USER CODE? WHY NOT JIT DEV CODE?

JITTING USER CODE? WHY NOT JIT DEV CODE?

- BREAKING DOWN THE WALLS OF THE COMPILER & LINKER

JITTING USER CODE? WHY NOT JIT DEV CODE?

- BREAKING DOWN THE WALLS OF THE COMPILER & LINKER
  - (+) INCREMENTAL COMPILATION ON A RUNNING SYSTEM

JITTING USER CODE? WHY NOT JIT DEV CODE?

- BREAKING DOWN THE WALLS OF THE COMPILER & LINKER
  - (+) INCREMENTAL COMPILATION ON A RUNNING SYSTEM
  - (+) PATCHING WITH SOURCE CODE

JITTING USER CODE? WHY NOT JIT DEV CODE?

- BREAKING DOWN THE WALLS OF THE COMPILER & LINKER
  - (+) INCREMENTAL COMPILATION ON A RUNNING SYSTEM
  - (+) PATCHING WITH SOURCE CODE
  - (+) PROGRAMMING WITH CONCEPTS: PLATFORM TEAM VS APPLICATION TEAM

SUMMARY / FUTURE?

[[CLANG::JIT]] - COSTS & BENEFITS

[[CLANG::JIT]] - COSTS & BENEFITS

(+) RUN-TIME PERFORMANCE (OPTIMIZATIONS BASED ON RUN-TIME DATA/AVAILABLE HARDWARE)

(-) BASED ON STRINGS (SECURITY CONCERNS)

[[CLANG::JIT]] - COSTS & BENEFITS

(+) RUN-TIME PERFORMANCE (OPTIMIZATIONS BASED ON RUN-TIME DATA/AVAILABLE HARDWARE)

(+) COMPILE-TIME PERFORMANCE (MEMOIZATION)

(-) BASED ON STRINGS (SECURITY CONCERNS)

(-) DOESN'T SUPPORT COMPILE-TIME INJECTION

[[CLANG::JIT]] - COSTS & BENEFITS

(+) RUN-TIME PERFORMANCE (OPTIMIZATIONS BASED ON RUN-TIME DATA/AVAILABLE HARDWARE)

(+) COMPILE-TIME PERFORMANCE (MEMOIZATION)

(+) DIRECT ACCESS TO THE HOST / STL ALGORITHMS FOR METAPROGRAMMING WITH TYPES

(-) BASED ON STRINGS (SECURITY CONCERNS)

(-) DOESN'T SUPPORT COMPILE-TIME INJECTION

FUTURE? / COMPILE-TIME & RUN-TIME

FUTURE? / COMPILE-TIME & RUN-TIME

```
template <class T>
auto future(); // No attribute at the declaration side
```

```
// Run-time
-> auto jit = "int"s;
   [[jit]] future<jit>() // Attribute at the calling side
```

FUTURE? / COMPILE-TIME & RUN-TIME

```
template <class T>
auto future(); // No attribute at the declaration side
```

```
// Run-time
-> auto jit = "int"s;
   [[jit]] future<jit>() // Attribute at the calling side
```

```
template <class T>
constexpr auto future();
```

```
// Run/Compile-time
-> auto jit = "int"s;
   [[jit]] future<jit>()
   [[jit]] future<"int">()
```

FUTURE? / COMPILE-TIME & RUN-TIME

```
template <class T>
auto future(); // No attribute at the declaration side
```

```
// Run-time
-> auto jit = "int"s;
   [[jit]] future<jit>() // Attribute at the calling side
```

```
template <class T>
constexpr auto future();
```

```
// Run/Compile-time
-> auto jit = "int"s;
   [[jit]] future<jit>()
   [[jit]] future<"int">()
```

```
template <class T>
consteval auto future();
```

```
// Compile-time
-> [[jit]] future<"int">()
```

FUTURE? / COMPILE-TIME & RUN-TIME

```
template <class T>
auto future(); // No attribute at the declaration side
```

```
 // Run-time
 -> auto jit = "int"s;
    [[jit]] future<jit>() // Attribute at the calling side
```

```
template <class T>
constexpr auto future();
```

```
 // Run/Compile-time
 -> auto jit = "int"s;
    [[jit]] future<jit>()
    [[jit]] future<"int">()
```

```
template <class T>
consteval auto future();
```

```
 // Compile-time
 -> [[jit]] future<"int">()
```

```
 -> future<int>() // Valid in all cases
```

FUTURE? / META - STRINGLESS SOLUTION

FUTURE? / META - STRINGLESS SOLUTION

```
auto i = 42; // Run-time value
```

FUTURE? / META - STRINGLESS SOLUTION

```
auto i = 42; // Run-time value
```

```
using type = reify(meta<std::integral_constant>(i)); // AST node
```

TO "REIFY" SOMETHING IS TO TAKE SOMETHING THAT IS ABSTRACT AND REGARD IT AS MATERIAL

P1717: COMPILETIME METAPROGRAMMING IN C++

FUTURE? / META - STRINGLESS SOLUTION

```
auto i = 42; // Run-time value
```

```
using type = reify(meta<std::integral_constant>(i)); // AST node
```

```
future<type>() // No attribute required
 // future<std::integral_constant<42>>()
```

TO "REIFY" SOMETHING IS TO TAKE SOMETHING THAT IS ABSTRACT AND REGARD IT AS MATERIAL

P1717: COMPILETIME METAPROGRAMMING IN C++

LET'S JIT ALL THE THINGS?!

QUESTIONS? / #SIG_JIT

HTTPS://GITHUB.COM/QUANTLABFINANCIAL | HTTPS://WWW.QUANTLAB.COM/CAREERS

BONUS SLIDES

JIT IN THE COMPILER - SPECULATION

JIT IN THE COMPILER - SPECULATION

Compilation is ~~becoming~~ quite complicated

JIT IN THE COMPILER - SPECULATION

Compilation is ~~becoming~~ quite complicated

P0992: TRANSLATION AND EVALUATION

JIT IN THE COMPILER

JIT IN THE COMPILER

- DATA PROMOTION FROM COMPILE-TIME TO RUNTIME?

JIT IN THE COMPILER

- DATA PROMOTION FROM COMPILE-TIME TO RUNTIME?
- COMPILATION TIME VARIABILITY?

JIT IN THE COMPILER

- DATA PROMOTION FROM COMPILE-TIME TO RUNTIME?
- COMPILATION TIME VARIABILITY?
- CROSS COMPILATION?