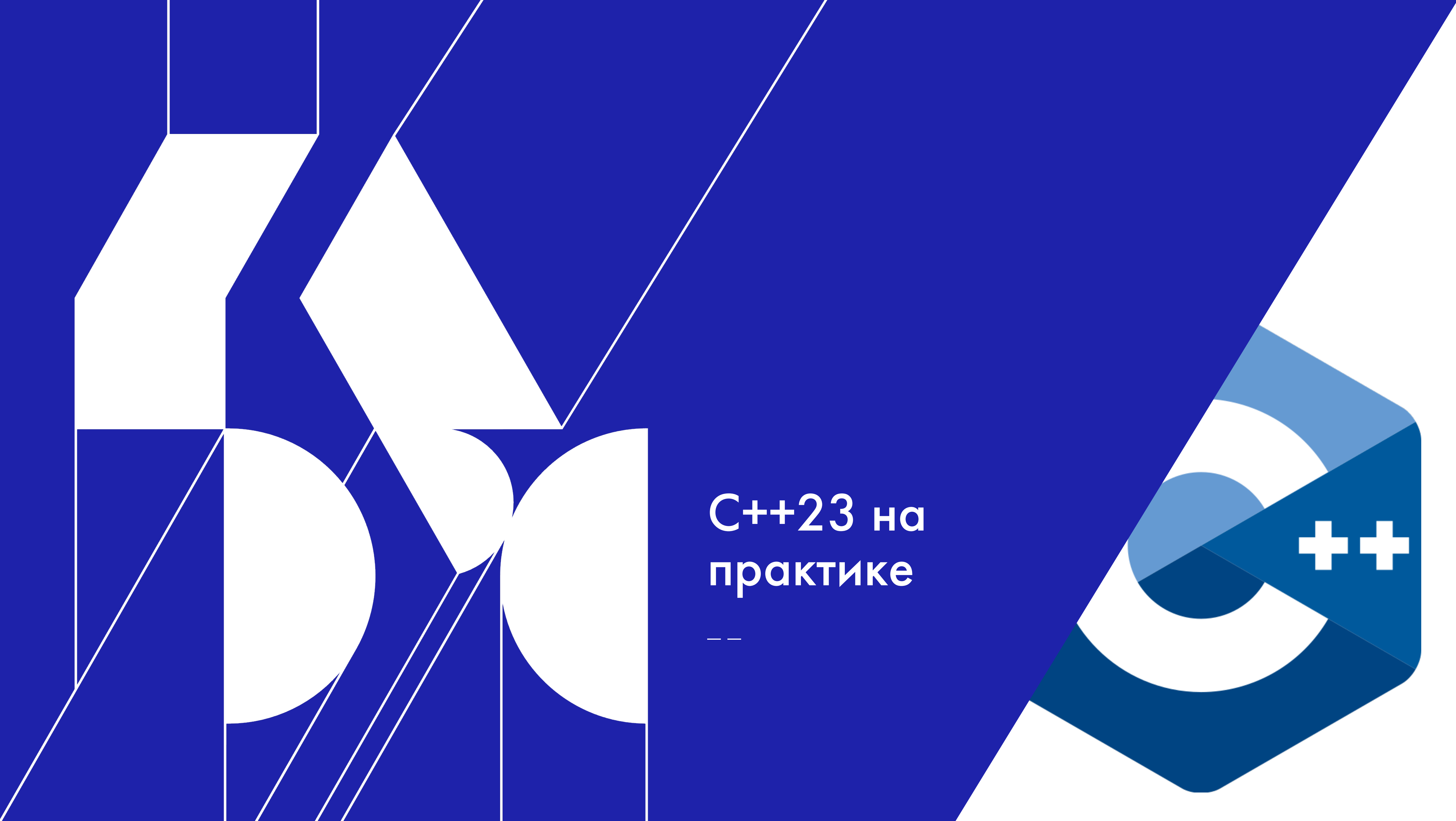
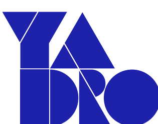
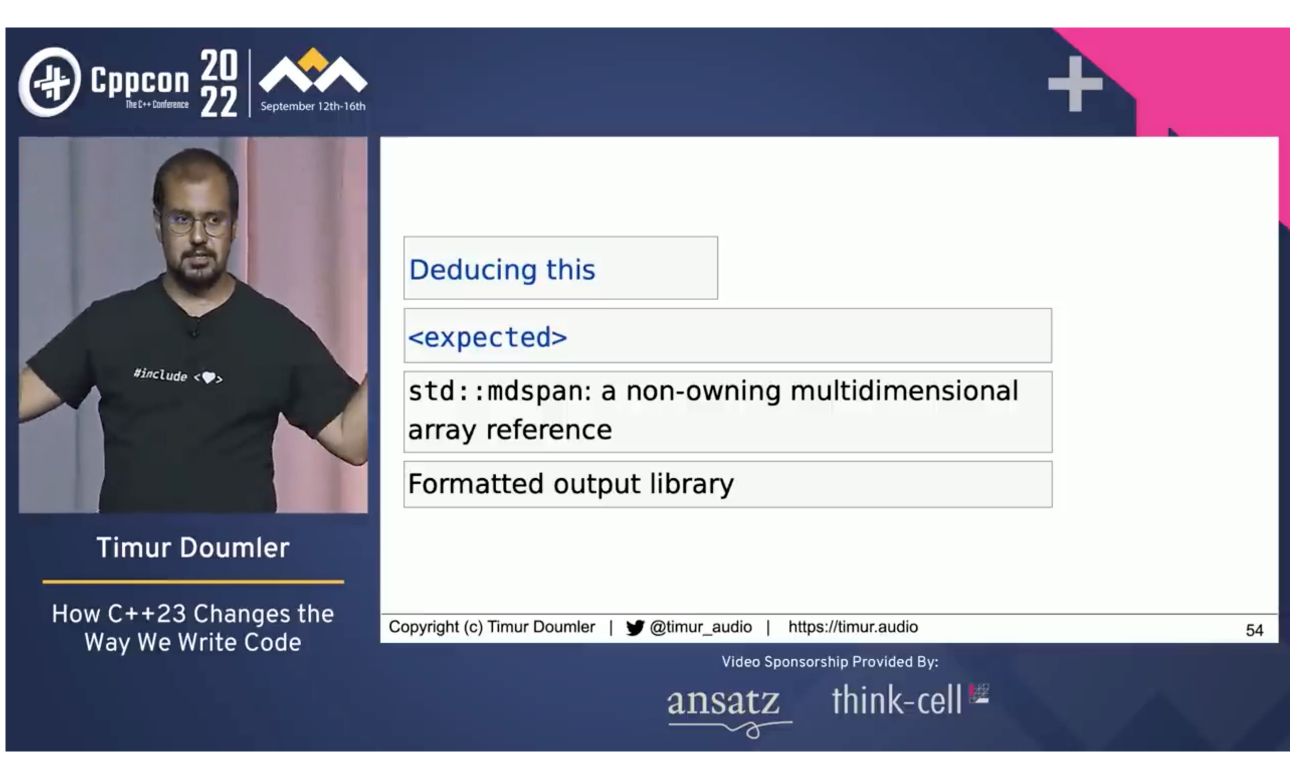
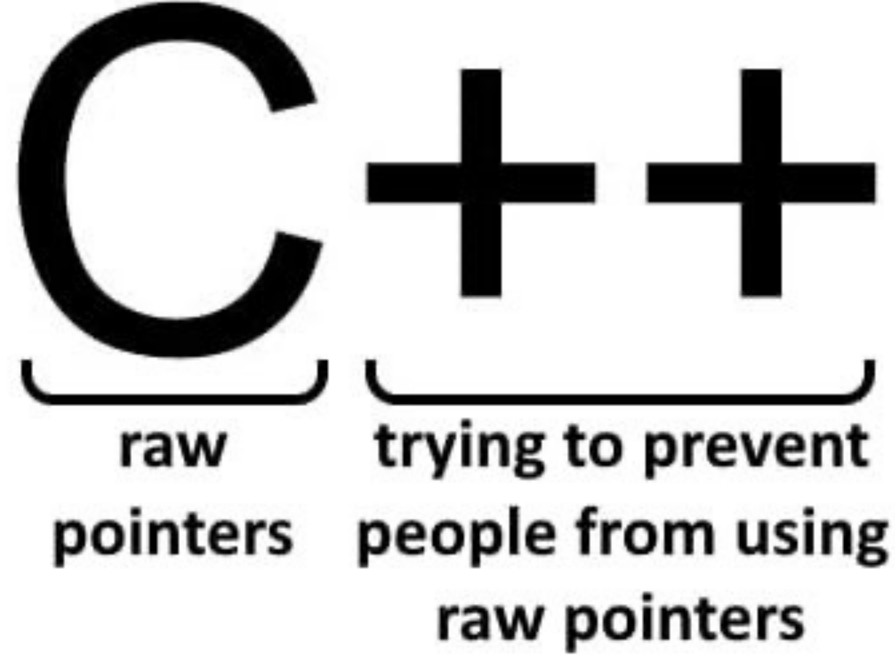

C++23 на практике
++

How C++23 Changes the Way We Write Code





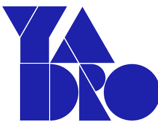

C++
raw
pointers
trying to prevent
people from using
raw pointers

```
class Ring {
        io_uring ring_;

        // ...
        io_uring_sqe* getSqe() noexcept {
                return ::io_uring_get_sqe(&ring_);
        }
        // ...
};
```

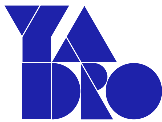
Избавляемся от сырых указателей в liburing

```
io_uring_sqe& getSqe() noexcept {
        return *::io_uring_get_sqe(&ring_);
}
```

```
std::optional<io_uring_sqe &> getSqe() noexcept {
      auto sqe = ::io_uring_get_sqe(&ring_);
      if (sqe) return *sqe;

      return std::nullopt;
}
```



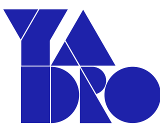
```
std::optional<std::reference_wrapper<io_uring_sqe>>
      getSqe() noexcept {
      auto sqe = ::io_uring_get_sqe(&ring_);
      if (sqe) return *sqe;

      return std::nullopt;
}
```




```cpp
using SQERefWrap = std::reference_wrapper<io_uring_sqe>;

std::optional<SQERefWrap> getSqe() noexcept {
      auto sqe = ::io_uring_get_sqe(&ring_);
      if (sqe) return *sqe;

      return std::nullopt;
}
```

```cpp
static std::optional<SQERefWrap> prepNop(io_uring_sqe&) noexcept;

auto sqe = getSqe();
if (sqe) prepNop(*sqe);
```

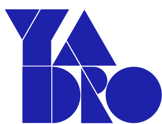
```
static std::optional<SQERefWrap> prepNop(io_uring_sqe&) noexcept;
io_uring_sqe* getSqe() noexcept;

auto sqe = getSqe();
if (sqe) prepNop(*sqe);
```



```
static std::optional<SQERefWrap> prepNop(io_uring_sqe&) noexcept;
std::optional<SQERefWrap> getSqe() noexcept;

auto sqe = getSqe();
if (sqe) prepNop(*sqe);
```

Избавляемся от сырых указателей в liburing

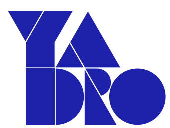
```
static std::optional<SQERefWrap> prepNop(io_uring_sqe&) noexcept;
std::optional<SQERefWrap> getSqe() noexcept;

getSqe()
      .and_then(prepNop);
```



```
static std::optional<SQERefWrap> prepNop(io_uring_sqe&) noexcept;
std::optional<SQERefWrap> getSqe() noexcept;

getSqe()
      .or_else([&] {
            ::io_uring_submit(&ring_);
            return getSqe();
      })
      .and_then(prepNop);
```

```
static std::optional<SQERefWrap> prepNop(io_uring_sqe&) noexcept;
std::optional<SQERefWrap> getSqe() noexcept;

getSqe()
        .or_else([&] {
                ::io_uring_submit(&ring_);
                return getSqe();
        })
        .and_then(prepNop)
        .transform([&](...) { return ::io_uring_submit(&ring_); });
```

Monadic std::optional



```cpp
std::optional<image> get_cute_cat(const image& img) {
      auto cropped = crop_to_cat(img);
      if (!cropped) return std::nullopt;

      auto with_tie = add_bow_tie(*cropped);
      if (!with_tie) return std::nullopt;

      auto with_sparkles = make_eyes_sparkle(*with_tie);
      if (!with_sparkles) return std::nullopt;

      return add_rainbow(make_smaller(*with_sparkles));
}
```

```
std::optional<image> get_cute_cat(const image& img) {
      return crop_to_cat(img)
             .and_then(add_bow_tie)
             .and_then(make_eyes_sparkle)
             .map(make_smaller)
             .map(add_rainbow);
}
```

Monadic std::optional

C++23 library features

C++23 feature	Paper(s)	GCC libstdc++	Clang libc++	MSVC STL	Apple Clang*	IBM Open XL C/C++ for AIX	Sun/Oracle C++*	Embarcadero C++ Builder*	[Collapse]
Monadic operations for `std::optional`	P0798R8	12	14	19.32*	14.0.3*				

Monadic std::optional

```
template< class F >
constexpr auto and_then( F&& f ) &;        (1)
```

```
template< class F >
constexpr auto transform( F&& f ) &;        (1)
```

```
template< class F >
constexpr optional or_else( F&& f ) const&;        (1)
```

Monadic std::optional

and_then/transform/or_else

1. and_then/transform/or_else возвращают новый std::optional

Monadic std::optional

and_then/transform/or_else

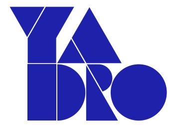
1. and_then/transform/or_else возвращают новый std::optional
2. and_then/transform/or_else имеют один аргумент – функция или Callable

Monadic std::optional

and_then/transform/or_else

1. and_then/transform/or_else возвращают новый std::optional
2. and_then/transform/or_else имеют один аргумент – функция или Callable
3. and_then принимает как аргумент функцию с одним аргументом – значением std::optional и возвращающую новый std::optional (не обязательно ту же специализацию)

Monadic std::optional



and_then/transform/or_else

1. and_then/transform/or_else возвращают новый std::optional
2. and_then/transform/or_else имеют один аргумент – функция или Callable
3. and_then принимает как аргумент функцию с одним аргументом – значением std::optional и возвращающую новый std::optional (не обязательно ту же специализацию)
4. transform принимает как аргумент функцию с одним аргументом – значением std::optional и возвращающую новое значение (кроме массивов, std::in_place_t или std::nullopt_t)

Monadic std::optional

and_then/transform/or_else

1. and_then/transform/or_else возвращают новый std::optional
2. and_then/transform/or_else имеют один аргумент – функция или Callable
3. and_then принимает как аргумент функцию с одним аргументом – значением std::optional и возвращающую новый std::optional (не обязательно ту же специализацию)
4. transform принимает как аргумент функцию с одним аргументом – значением std::optional и возвращающую новое значение (кроме массивов, std::in_place_t или std::nullopt_t)
5. or_else принимает как аргумент функцию без аргументов и возвращающую новый std::optional от того же типа

```
class Descriptor {
      int fd_;
};
```

```
class Descriptor {
      int fd_;
public:
      explicit Descriptor(const char* path, int flags);
      ~Descriptor() { ::close(fd_); }
};
```

Больше обёрток богу обёрток

```
class Descriptor {
      int fd_;
public:
      explicit Descriptor(const char* path, int flags);
      ~Descriptor() { ::close(fd_); }
      Descriptor(const Descriptor&) = delete;
      Descriptor& operator=(const Descriptor&) = delete;
};
```

```cpp
class Descriptor {
      int fd_;
public:
      explicit Descriptor(const char* path, int flags);
      ~Descriptor() { ::close(fd_); }
      Descriptor(const Descriptor&) = delete;
      Descriptor& operator=(const Descriptor&) = delete;
      Descriptor(Descriptor&&) noexcept;
      Descriptor& operator=(Descriptor&&) noexcept;
};
```

Больше обёрток богу обёрток

```
class Descriptor {
        int fd_;
public:
        explicit Descriptor(const char* path, int flags);
        ~Descriptor() { ::close(fd_); }
        Descriptor(const Descriptor&) = delete;
        Descriptor& operator=(const Descriptor&) = delete;
        Descriptor(Descriptor&&) noexcept;
        Descriptor& operator=(Descriptor&&) noexcept;
        auto value() const noexcept { return fd_; }
};
```

```
Descriptor::Descriptor(const char* path, int flags)
      : fd_(::open(path, flags)) {

      if (fd_ < 0) throw std::system_error{
                                  errno,
                                  std::generic_category(),
                                  path};
}
```

```cpp
class Descriptor {
        // ...
public:
        static Descriptor create(
                const char* path, int flags) noexcept;
        // ...
private:
        Descriptor(int fd) : fd_(fd) noexcept {}
};
```

```
Descriptor Descriptor::create(
      const char* path, int flags) noexcept {

      auto fd = ::open(path, flags);
      if (fd >= 0) return Descriptor{fd};

      ???
}
```

```
Descriptor | err Descriptor::create(
        const char* path, int flags) noexcept {

        auto fd = ::open(path, flags);
        if (fd >= 0) return Descriptor{fd};

        return errno;
}
```

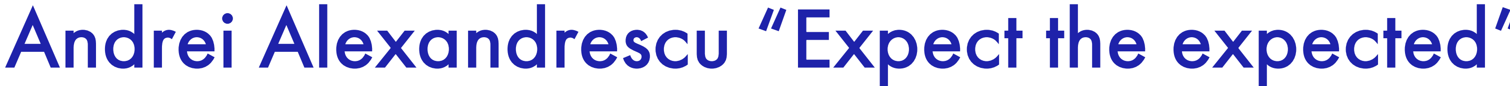
```
struct CreateResult {
	union {
		Descriptor desc;
		int err;
	};
	bool ok;
};
```

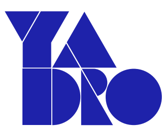
```
CreateResult Descriptor::create(
        const char* path, int flags) noexcept {

        auto fd = ::open(path, flags);
        if (fd >= 0)
                return {
                        .desc = Descriptor{fd},
                        .ok = true,
                };

        return {
                .err = errno,
                .ok = false,
        };
}
```

```cpp
std::variant<Descriptor, int> Descriptor::create(
      const char* path, int flags) noexcept {

      auto fd = ::open(path, flags);
      if (fd >= 0) return Descriptor{fd};

      return errno;
}
```

```
std::expected<Descriptor, int> Descriptor::create(
      const char* path, int flags) noexcept {

      auto fd = ::open(path, flags);
      if (fd >= 0) return Descriptor{fd};

      return std::unexpected{errno};
}
```

std::expected





Implementation (partial)

```
template<class T, class E> class expected {
   union { T yay; E nay; };
   bool ok = true;
public:
   expected() { new(&yay) T(); }
   expected(const T& rhs) { new(&yay) T(rhs); }
   expected(const unexpected<E>& rhs) : ok(false) {
      new(&nay) E(rhs.value());
   }
   template<class U = T> explicit expected(U&& rhs) {
      new(&yay) T(forward<U>(rhs));
   }
   ...
};
```

24 / 36

cppcon | 2018

THE C++ CONFERENCE • BELLEVUE, WASHINGTON

ANDREI ALEXANDRESCU

Expect the expected

CppCon.org

std::expected

C++23 library features

C++23 feature	Paper(s)	GCC libstdc++	Clang libc++	MSVC STL	Apple Clang*	IBM Open XL C/C++ for AIX*	Sun/Oracle C++*	Embarcadero C++ Builder*	[Collapse]
<expected>	P0323R12 P2549R1	12	16	19.33*					

std::expected

Observers

`operator->` `operator*`	accesses the expected value (public member function)
`operator bool` `has_value`	checks whether the object contains an expected value (public member function)
`value`	returns the expected value (public member function)
`error`	returns the unexpected value (public member function)
`value_or`	returns the expected value if present, another value otherwise (public member function)

std::expected

Monadic operations

`and_then`	returns the result of the given function on the expected value if it exists; otherwise, returns the `expected` itself (public member function)
`transform`	returns an `expected` containing the transformed expected value if it exists; otherwise, returns the `expected` itself (public member function)
`or_else`	returns the `expected` itself if it contains an expected value; otherwise, returns the result of the given function on the unexpected value (public member function)
`transform_error`	returns the `expected` itself if it contains an expected value; otherwise, returns an `expected` containing the transformed unexpected value (public member function)

Monadic std::expected

- and_then
- transform
- or_else
- transform_error

Monadic std::expected and_then

1. Принимает как аргумент функтор
2. Если T = void функтор не должен принимать аргументы
3. Если T != void функтор должен принимать как аргумент значение из expected
4. Функтор должен возвращать новый expected
5. Возвращаемый expected может иметь новый тип значения
6. Возвращаемый expected должен иметь тот же самый тип ошибки

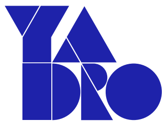
Monadic std::expected transform

1. Принимает как аргумент функтор
2. Если T = void функтор не должен принимать аргументы
3. Если T != void функтор должен принимать как аргумент значение из expected
4. Функтор может возвращать почти любой тип, даже void
5. Функтор не должен возвращать std::unexpected, std::unexpect_t, std::in_place_t, reference
6. Возвращаемое значение обернется в новый expected с тем же типом ошибки

Monadic std::expected or_else

1. Принимает как аргумент функтор
2. Функтор всегда должен всегда принимать один аргумент – ошибку
3. Функтор должен возвращать новый expected
4. Возвращаемый expected может иметь новый тип ошибки
5. Возвращаемый expected должен иметь тот же самый тип значения

1. Принимает как аргумент функтор
2. Функтор всегда должен всегда принимать один аргумент – ошибку
3. Функтор может возвращать почти любой тип
4. Возвращаемое значение не может быть void и unexpected
5. Возвращаемое значение обернется в новый expected с тем же типом значения



/dev/scst_user дескриптор

1. Работа с /dev/scst_user через ioctl
2. При открытии дескриптора создаются ядровые структуры на которые можно посмотреть через sysfs
3. Структуры должны быть удалены после закрытия дескриптора

```
TEST(Foo, CreateDisk1) {
        auto dev = scst::Dev{getScstUserDevDesc()};

        ASSERT_TRUE(std::filesystem::exists(getPathToSysFs()));
}

TEST(Foo, CreateDisk2) {
        auto dev = scst::Dev{getScstUserDevDesc()};

        ASSERT_TRUE(std::filesystem::exists(getPathToSysFs()));
}
```

scst

```
[==========] Running 2 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 2 tests from Foo
[ RUN      ] Foo.CreateDisk1
[       OK ] Foo.CreateDisk1 (1 ms)
[ RUN      ] Foo.CreateDisk2
unknown file: Failure
C++ exception with description "Error registering SCST device: Cannot allocate memory" thrown in the test body.
[  FAILED  ] Foo.CreateDisk2 (0 ms)
[----------] 2 tests from Foo (1 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (1 ms total)
[  PASSED  ] 1 test.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] Foo.CreateDisk2

 1 FAILED TEST
```

scst

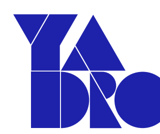
```
[95135]: scst: Virtual device handler TestDev for type 0 registered successfully
[95135]: scst_user: Attached user space virtual device "TestDev"
[95135]: scst: Attached to virtual device TestDev (id 1)
[95136]: scst_user: Releasing dev TestDev
[95136]: scst: Detached from virtual device TestDev (id 1)
[95136]: scst_user: Detached user space virtual device "TestDev"
[95136]: scst: Device handler "TestDev" unloaded
[95135]: scst: Virtual device handler TestDev for type 0 registered successfully
[95135]: scst_user: Attached user space virtual device "TestDev"
[95135]: scst: Attached to virtual device TestDev (id 2)
[95137]: scst_user: Releasing dev TestDev
[95137]: scst: Detached from virtual device TestDev (id 2)
[95137]: scst_user: Detached user space virtual device "TestDev"
[95137]: scst: Device handler "TestDev" unloaded
[95414]: scst: Virtual device handler TestDev for type 0 registered successfully
[95414]: scst_user: Attached user space virtual device "TestDev"
[95414]: scst: Attached to virtual device TestDev (id 3)
[95415]: scst_user: Releasing dev TestDev
[95415]: scst: Detached from virtual device TestDev (id 3)
[95414]: scst: ***ERROR***: SGV pool TestDevSgv already exists
[95415]: scst_user: Detached user space virtual device "TestDev"
[95415]: scst: Device handler "TestDev" unloaded
```

```cpp
class WaitWrapper {
	std::shared_ptr<IWaitStrategy> wait_;
public:
	~WaitWrapper() {
		wait_->wait();
	}

	int value() const noexcept;
};
```

```
class WaitWrapper {
      std::shared_ptr<IWaitStrategy> wait_;
      std::aligned_storage_t<sizeof(Descriptor)> desc_;
public:
      ~WaitWrapper() {
            std::destroy_at(reinterpret_cast<Descriptor*>(&desc_));
            wait_->wait();
      }

      auto value() const noexcept;
};
```

```
class WaitWrapper {
      std::shared_ptr<IWaitStrategy> wait_;
      std::aligned_storage_t<sizeof(Descriptor)> desc_;
public:
      ~WaitWrapper() {
            std::destroy_at(reinterpret_cast<Descriptor*>(&desc_));
            wait_->wait();
      }

      auto value() const noexcept;
};
```

deprecation of aligned_storage/aligned_union



```
template <typename T>
using aligned_storage = alignas(T) std::byte[sizeof(T)];
```

```
class WaitWrapper {
        std::shared_ptr<IWaitStrategy> wait_;
        alignas(Descriptor) std::byte desc_[sizeof(Descriptor)];
public:
        ~WaitWrapper() {
                std::destroy_at(reinterpret_cast<Descriptor*>(&desc_));
                wait_->wait();
        }

        auto value() const noexcept;
};
```

Заключение

1. Монадический std::optional – прикольно

Заключение

1. Монадический std::optional – прикольно
2. std::expected – еще прикольней

Заключение

1. Монадический std::optional – прикольно
2. std::expected – еще прикольней
3. aligned_storage/aligned_union – я буду помнить вас всегда