Projektowanie złożonych systemów telekomunikacyjnych

Modern C++:
Optional, variant, lambdas

Aleksander Miera

Agenda

1. A bit of context: error handling
2. Variant
3. A short interlude: visitor pattern
4. Lambdas and function objects
5. Optional
6. Advanced topics (also beyond std::)

Error handling

```
struct ConnectedTcpClient
{
    std::unique_ptr<ConnectedTcpSocket> socket{};
    std::vector<std::byte> receiveOneKb()
    {
        static constexpr auto kilobyteBytes=1024u;
        std::vector<std::byte> rxdata(kilobyteBytes);
        auto nreceived = socket->receive(rxdata.data(), kilobyteBytes);
        if (nreceived < 0) {
            if (errno == EWOULDBLOCK)

            //handle error somehow?
        }
        if (nreceived < kilobyteBytes) {
            rxdata.resize(static_cast<std::size_t>(nreceived));
        }
        return rxdata;
    }
};
```

Classical POSIX/C way of handling errors

Error handling

```
struct ConnectedTcpClient
{
    std::unique_ptr<ConnectedTcpSocket> socket{};
    std::vector<std::byte> receiveOneKb()
    {
        static constexpr auto kilobyteBytes=1024u;
        std::vector<std::byte> rxdata(kilobyteBytes);
        auto nreceived = socket->receive(rxdata.data(), kilobyteBytes);
        if (nreceived < 0) {
            if (errno == EWOULDBLOCK)
                throw std::runtime_error("recv: EWOULDBLOCK");
            //handle error somehow?
        }
        if (nreceived < kilobyteBytes) {
            rxdata.resize(static_cast<std::size_t>(nreceived));
        }
        return rxdata;
    }
};
```

This works

https://godbolt.org/z/r8KvsrPvn

Error handling

```
struct ConnectedTcpClient
{
    std::unique_ptr<ConnectedTcpSocket> socket{};
    std::vector<std::byte> receiveOneKb()
    {
        static constexpr auto kilobyteBytes=1024u;
        std::vector<std::byte> rxdata(kilobyteBytes);
        auto nreceived = socket->receive(rxdata.data(), kilobyteBytes);
        if (nreceived < 0) {
            if (errno == EWOULDBLOCK)
                throw std::runtime_error("recv: EWOULDBLOCK");
            //handle error somehow?
        }
        if (nreceived < kilobyteBytes) {
            rxdata.resize(static_cast<std::size_t>(nreceived));
        }
        return rxdata;
    }
};
```

This works
...or does it?

NOKIA

Error handling

```
int foo() noexcept;
int bar() noexcept(true);
int baz();
int faz() noexcept(false);
```

Error handling

```
int foo() noexcept;
int bar() noexcept(true);
int baz();
int faz() noexcept(false);
```

This is guaranteed not to throw

Error handling

```
int foo() noexcept;
int bar() noexcept(true);
int baz();
int faz() noexcept(false);
```

This is guaranteed not to throw

So is this

Error handling

```
int foo() noexcept;
int bar() noexcept(true);
int baz();
int faz() noexcept(false);
```

This is guaranteed not to throw

So is this

This is allowed to throw

Error handling

```
int foo() noexcept;
int bar() noexcept(true);
int baz();
int faz() noexcept(false);
```

This is guaranteed not to throw

So is this

This is allowed to throw
(or the author forgot to care ☹)

Error handling

```
int foo() noexcept;
int bar() noexcept(true);
int baz();
int faz() noexcept(false);
```

This is guaranteed not to throw

So is this

This is allowed to throw
(or the author forgot to care ☹)

This is allowed to throw, too,
but the author was explicit about it

Error handling

```
int foo() noexcept;
int bar() noexcept(true);
int baz();
int faz() noexcept(false);
```

This is guaranteed not to throw

So is this

This is allowed to throw
(or the author forgot to care ☹)

This is allowed to throw, too,
but the author was explicit about it
(be grateful to that person, **really**)

Error handling

```
int baz();
int faz() noexcept(false);
```

OK, so what kind of exceptions is this allowed to throw?

Error handling

```
int baz();
int faz() noexcept(false);
```

OK, so what kind of exceptions is this allowed to throw?

?

Error handling

```
int baz();
int faz() noexcept(false);
```

OK, so what kind of exceptions is this allowed to throw?

? ?

Error handling

```
int baz();
int faz() noexcept(false);
```

OK, so what kind of exceptions is this allowed to throw?

? ? ?

Error handling

```
int baz();
int faz() noexcept(false);
```

OK, so what kind of exceptions is this allowed to throw?

ANY!

Error handling

```cpp
std::vector<std::byte> receiveOneKb()
{
    static constexpr auto kilobyteBytes=1024u;
    std::vector<std::byte> rxdata(kilobyteBytes);
    auto nreceived = socket->receive(rxdata.data(), kilobyteBytes);
    if (nreceived < 0) {
        if (errno == EWOULDBLOCK)
            throw 0xDEADBEEF;
        //handle error?
    }
    if (nreceived < kilobyteBytes) {
        rxdata.resize(static_cast<std::size_t>(nreceived));
    }
    return rxdata;
}
```

Mind the function prototype

https://godbolt.org/z/bTdneTxcq

Error handling

```
std::vector<std::byte> receiveOneKb()
{
    static constexpr auto kilobyteBytes=1024u;
    std::vector<std::byte> rxdata(kilobyteBytes);
    auto nreceived = socket->receive(rxdata.data(), kilobyteBytes);
    if (nreceived < 0) {
        if (errno == EWOULDBLOCK)
            throw 0xDEADBEEF;
        //handle error?
    }
    if (nreceived < kilobyteBytes) {
        rxdata.resize(static_cast<std::size_t>(nreceived));
    }
    return rxdata;
}
```

Mind the function prototype

Throw an int? Sure, why not?

https://godbolt.org/z/bTdneTxcq

Error handling

```
std::vector<std::byte> receiveOneKb()
{
    static constexpr auto kilobyteBytes=1024u;
    std::vector<std::byte> rxdata(kilobyteBytes);
    auto nreceived = socket->receive(rxdata.data(), kilobyteBytes);
    if (nreceived < 0) {
        if (errno == EWOULDBLOCK)
            throw "errno EWOULDBLOCK";
        //handle error?
    }
    if (nreceived < kilobyteBytes) {
        rxdata.resize(static_cast<std::size_t>(nreceived));
    }
    return rxdata;
}
```

Mind the function prototype

C-string? Be my guest

https://godbolt.org/z/6zcr9rhjE

Error handling

Both previous snippets are examples can be considered bad code

Error handling

Both previous snippets are examples can be considered bad code

...but code like this exists in production.

Error handling

Both previous snippets are examples can be considered bad code

...but code like this exists in production.

Side note: older C++ had dynamic exception specification, but it was deprecated
https://en.cppreference.com/w/cpp/language/except_spec

Error handling

What else can be done for exception-less error handling?

Error handling

C-style error codes+output arguments:

```
int receiveOneKb(std::vector<std::byte>&)
```


Error handling

C-style error codes+output arguments:

```
int receiveOneKb(std::vector<std::byte>&)
```

1. Works, but requires massive number of if statements

Error handling

C-style error codes+output arguments:

```cpp
int receiveOneKb(std::vector<std::byte>&)
```

1. Works, but requires massive number of if statements
2. What is the output arguments is not default-constructible?

Error handling

C-style error codes+output arguments:

```
int receiveOneKb(std::vector<std::byte>&)
```

1. Works, but requires massive number of if statements
2. What is the output arguments is not default-constructible?
3. If error needs handling someplace higher-up the callstack, the interface might propagate

Error handling

Tagged union of error code and payload -- active member depends on the result

```
union RxResultImpl
{
    std::vector<std::byte> payload;
    int errorCode;
};

struct RxResult
{
    bool isOk;
    RxResultImpl data;
};

RxResult receiveOneKb()
```


Error handling

Tagged union of error code and payload -- active member depends on the result

```
union RxResultImpl
{
    std::vector<std::byte> payload;
    int errorCode;
};

struct RxResult
{
    bool isOk;
    RxResultImpl data;
};

RxResult receiveOneKb()
```

Seems better, but unions are pain to manage in C++ ☹

Error handling

Tagged union of error code and payload -- active member depends on the result

Error handling

Tagged union of error code and payload -- active member depends on the result but fully C++ compliant

Error handling

Tagged union of error code and payload -- active member depends on the result but fully C++ compliant

```
std::variant<std::vector<std::byte>, int> receiveOneKb()
```


Error handling

```
std::variant<std::vector<std::byte>, int> receiveOneKb() noexcept
{
    static constexpr auto kilobyteBytes=1024u;
    std::vector<std::byte> rxdata(kilobyteBytes);
    auto nreceived = socket->receive(rxdata.data(), kilobyteBytes);
    if (nreceived < 0) {
        return errno;
    }
    if (nreceived < kilobyteBytes) {
        rxdata.resize(static_cast<std::size_t>(nreceived));
    }
    return rxdata;
}
```

https://godbolt.org/z/MhTK4W9Te

Error handling

```
std::variant<std::vector<std::byte>, int> receiveOneKb() noexcept
{
    static constexpr auto kilobyteBytes=1024u;
    std::vector<std::byte> rxdata(kilobyteBytes);
    auto nreceived = socket->receive(rxdata.data(), kilobyteBytes);
    if (nreceived < 0) {
        return errno;
    }
    if (nreceived < kilobyteBytes) {
        rxdata.resize(static_cast<std::size_t>(nreceived));
    }
    return rxdata;
}
```

https://godbolt.org/z/MhTK4W9Te

Variant

```cpp
std::variant<char, double, int, float> varNumberType{};
```

List of types to be held (any number)

Constructor parameters

Variant

```
std::variant<char, double, int, float> varNumberType{};
```

List of types to be held (any number)

Constructor parameters

```
std::cout << std::holds_alternative<char>(varNumberType) << '\n';
```

True or false?

Variant

```
std::variant<char, double, int, float> varNumberType{};
```

List of types to be held (any number)

Constructor parameters

```
std::cout << std::holds_alternative<char>(varNumberType) << '\n';
```

True or false?
True. **Default construction constructs the first type**

https://godbolt.org/z/K68Meqqfv

Variant

```cpp
std::variant<char, double, int, float> varNumberType{};
```

List of types to be held (any number)

Constructor parameters

```cpp
std::cout << std::holds_alternative<char>(varNumberType) << '\n';
```

True or false?
True. **Default construction constructs the first type**

Variant

```
std::variant<char, double, int, float> varNumberType{3.14f};
std::cout << std::holds_alternative<     >(varNumberType) << '\n';
```

Variant

```
std::variant<char, double, int, float> varNumberType{3.14f};
std::cout << std::holds_alternative<float>(varNumberType) << '\n';
```

https://godbolt.org/z/7rKorYTnn

Variant

```
struct NotDefaultConstructible
{
    NotDefaultConstructible(int){}
};

std::variant<NotDefaultConstructible, float> v{};
```

Fails to compile, now what?

https://godbolt.org/z/5rbKqW1ha

Variant

```
struct NotDefaultConstructible
{
    NotDefaultConstructible(int){}
};

std::variant<NotDefaultConstructible, float> v{1};
```

1. Provide default constructor to the `NotDefaultConstructible`

https://godbolt.org/z/86jfcT1br

Variant

```
struct NotDefaultConstructible
{
    NotDefaultConstructible(int){}
};

std::variant<NotDefaultConstructible, float> v{1};
```

1. Provide default constructor to the `NotDefaultConstructible`
   ...but its construction needs to be deferred!

Variant

```cpp
struct NotDefaultConstructible
{
    NotDefaultConstructible(int){}
};

std::variant<std::monostate, NotDefaultConstructible, float> v{};
```

1. Provide default constructor to the `NotDefaultConstructible`
   ...but its construction needs to be deferred!
2. Use std::monostate

Variant

```
struct NotDefaultConstructible
{
    NotDefaultConstructible(int){}
};

std::variant<std::monostate, NotDefaultConstructible, float> v{};
```

1. Provide default constructor to the `NotDefaultConstructible`
   ...but its construction needs to be deferred!
2. Use std::monostate
   ... at the price of expanding the variant with one more type

https://godbolt.org/z/jnbefK7z1

Variant

```
struct NotDefaultConstructible
{
    NotDefaultConstructible(int){}
};

std::variant<std::monostate, NotDefaultConstructible, float> v{};
```

1. Provide default constructor to the `NotDefaultConstructible`
   ...but its construction needs to be deferred!
2. Use std::monostate
   ... at the price of expanding the variant with one more type

https://godbolt.org/z/jnbefK7z1

Variant

```
using VarNum = std::variant<double, int, float, char>;
VarNum douglas{42};
VarNum adams{42};
VarNum cadams{char(42)};
VarNum pi{3.14f};

std::cout << (douglas == adams) << '\n';

std::cout << (douglas == cadams) << '\n';

std::cout << (pi < cadams) << '\n';

std::cout << (pi < douglas) << '\n';
```


Variant

```
using VarNum = std::variant<double, int, float, char>;
VarNum douglas{42};
VarNum adams{42};
VarNum cadams{char(42)};
VarNum pi{3.14f};

std::cout << (douglas == adams) << '\n';
std::cout << (douglas == cadams) << '\n';
std::cout << (pi < cadams) << '\n';
std::cout << (pi < douglas) << '\n';
```

True, same value, same type.
Intuitive

Variant

```
using VarNum = std::variant<double, int, float, char>;
VarNum douglas{42};
VarNum adams{42};
VarNum cadams{char(42)};
VarNum pi{3.14f};

std::cout << (douglas == adams) << '\n';
std::cout << (douglas == cadams) << '\n';
std::cout << (pi < cadams) << '\n';
std::cout << (pi < douglas) << '\n';
```

True, same value, same type.
Intuitive

False, same value, different types.
Rather intuitive

Variant

```
using VarNum = std::variant<double, int, float, char>;
VarNum douglas{42};
VarNum adams{42};
VarNum cadams{char(42)};
VarNum pi{3.14f};

std::cout << (douglas == adams) << '\n';
std::cout << (douglas == cadams) << '\n';
std::cout << (pi < cadams) << '\n';
std::cout << (pi < douglas) << '\n';
```

True, same value, same type. Intuitive

False, same value, different types. Rather intuitive

True, left hand side smaller. Different types. Probably intuitive

Variant

```
using VarNum = std::variant<double, int, float, char>;
VarNum douglas{42};
VarNum adams{42};
VarNum cadams{char(42)};
VarNum pi{3.14f};

std::cout << (douglas == adams) << '\n';
std::cout << (douglas == cadams) << '\n';
std::cout << (pi < cadams) << '\n';
std::cout << (pi < douglas) << '\n';
```

True, same value, same type. Intuitive

False, same value, different types. Rather intuitive

True, left hand side smaller. Different types. Probably intuitive

False. WAT?

https://godbolt.org/z/feToGT7dW

Variant

If types are the same, normal comparison occurs. Otherwise type indices are compared

1-7) Compares two `std::variant` objects `lhs` and `rhs`. The contained values are compared (using the corresponding operator of `T`) only if both `lhs` and `rhs` contain values corresponding to the same index. Otherwise,

- `lhs` is considered *equal to* `rhs` if, and only if, both `lhs` and `rhs` do not contain a value.
- `lhs` is considered *less than* `rhs` if, and only if, either `rhs` contains a value and `lhs` does not, or `lhs.index()` is less than `rhs.index()`.

https://en.cppreference.com/w/cpp/utility/variant/operator_cmp

Variant

So, how to handle it in a safer manner?

Variant

So, how to handle it in a safer manner?

`std::holds alternative` seems a bit impractical

Variant

So, how to handle it in a safer manner?

```
std::cout << (std::get<int>(douglas) < std::get<double>(pi))
<< '\n' ;
```


Variant

So, how to handle it in a safer manner?

```
std::cout << (std::get<int>(douglas) < std::get<double>(pi))
<< '\n' ;
```

OK, that works, provided we know the types inside the variants or are ready to catch exceptions

https://godbolt.org/z/6fv6zfa55

Variant

So, how to handle it in a safer manner?

```
std::cout << (std::get<int>(douglas) < std::get<double>(pi)) << '\n' ;
```

OK, that works, provided we know the types inside the variants or are ready to catch exceptions in case we're wrong

```
std::cout << (std::get<double>(douglas) < std::get<double>(pi)) << '\n' ;
```

```
terminate called after throwing an instance of 'std::bad_variant_access'
  what():  std::get: wrong index for variant
Program terminated with signal: SIGSEGV
```

https://godbolt.org/z/M8h3hnEGT

Variant

OK, so maybe by index?

```
std::cout << (std::get<1>(douglas) < std::get<0>(pi)) << '\n' ;
```

OK, that works, provided we know the types inside the variants or are ready to catch exceptions in case we're wrong

```
terminate called after throwing an instance of 'std::bad_variant_access'
  what():  std::get: wrong index for variant
Program terminated with signal: SIGSEGV
```

https://godbolt.org/z/M8h3hnEGT

Variant

OK, so maybe by index?

```
std::cout << (std::get<1>(douglas) < std::get<0>(pi)) << '\n' ;
```

```
terminate called after throwing an instance of 'std::bad_variant_access'
  what():  std::get: wrong index for variant
Program terminated with signal: SIGSEGV
```


Variant

OK, so maybe by index?

```
std::cout << (std::get<1>(douglas) < std::get<0>(pi)) << '\n' ;
```

```
terminate called after throwing an instance of 'std::bad_variant_access'
  what():  std::get: wrong index for variant
Program terminated with signal: SIGSEGV
```

Oh, snap, it was float, not double ☹

https://godbolt.org/z/va1GEjc1a

Variant

OK, so maybe by index?

```
std::cout << (std::get<1>(douglas) < std::get<0>(pi)) << '\n' ;
```

```
terminate called after throwing an instance of 'std::bad_variant_access'
  what():  std::get: wrong index for variant
Program terminated with signal: SIGSEGV
```

Oh, snap, it was float, not double ☹

https://godbolt.org/z/va1GEjc1a

Variant

OK, so maybe by index?

```
std::cout << (std::get<1>(douglas) < std::get<2>(pi)) << '\n' ;
```

Variant

OK, so maybe by index?

```
std::cout << (std::get<1>(douglas) < std::get<2>(pi)) << '\n' ;
```

OK, now it works, but it still throws and is practial only when we know what’s inside upfront

https://godbolt.org/z/dWE9YzeGP

Variant

What else can be done?

```
const int* douglas_ = std::get_if<int>(&douglas);
const float* pi_ = std::get_if<float>(&pi);

if (douglas_ && pi_) {
    std::cout << (*douglas_ < *pi_) << '\n' ;
}
```

Variant

What else can be done?

```
const int* douglas_ = std::get_if<int>(&douglas);
const float* pi_ = std::get_if<float>(&pi);

if (douglas_ && pi_) {
    std::cout << (*douglas_ < *pi_) << '\n' ;
}
```

Makes sense when checking a single variable for a single type,
but impractical for multiple variants/contained types; make the code C-style.

https://godbolt.org/z/bYo7PGdez

Variant

Can this be done in a **generic** manner?

Variant

Can this be done in a **generic** manner?

Visitor pattern to the rescue!

Variant

Can this be done in a **generic** manner?

Visitor pattern to the rescue!

Variant

```
template< class R, class Visitor, class... Variants >
constexpr R visit( Visitor&& v, Variants&&... values );
```

(2) (since C++20)

Variant

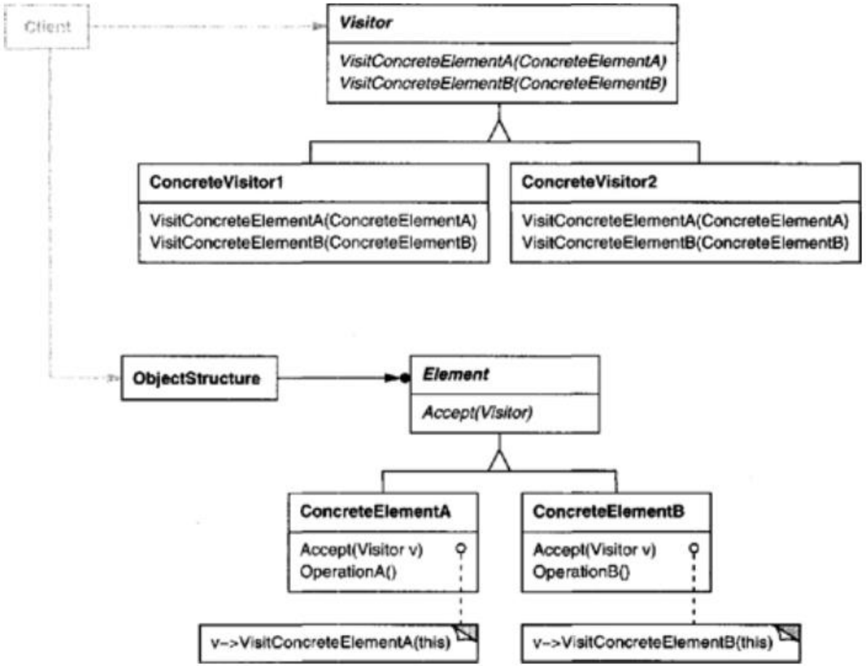
```
template< class R, class Visitor, class... Variants >
constexpr R visit( Visitor&& v, Variants&&... values );
```
(2) (since C++20)

```
auto lowerThanCompare = [](auto l, auto r) {
    return l < r;
};

std::cout << std::visit(lowerThanCompare, douglas, pi) << '\n' ;
```

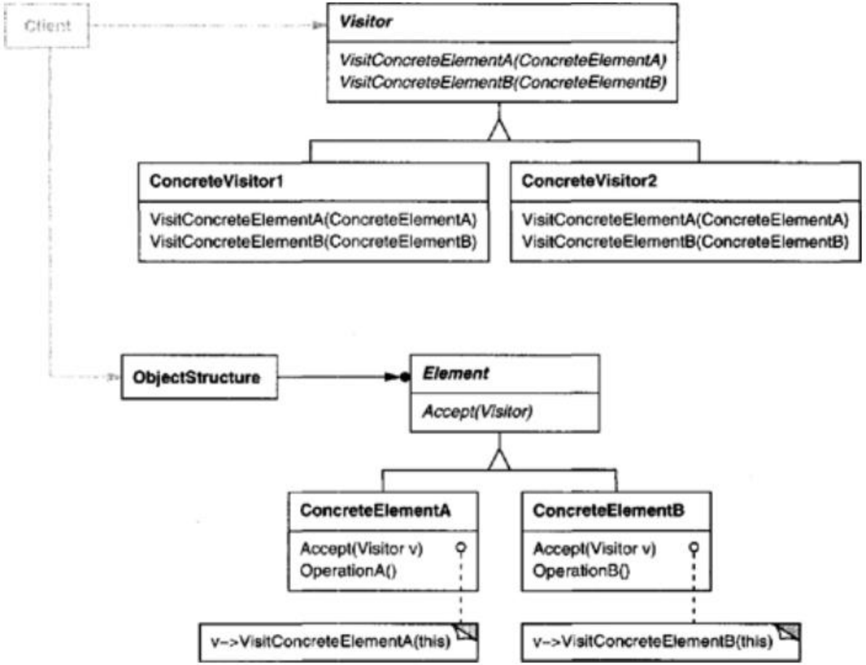
https://godbolt.org/z/8WG3v9M3r

Visitor



Client
Visitor
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor1
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor2
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ObjectStructure
Element
Accept(Visitor)
ConcreteElementA
Accept(Visitor v)
OperationA()
ConcreteElementB
Accept(Visitor v)
OperationB()
v->VisitConcreteElementA(this)
v->VisitConcreteElementB(this)

Visitor

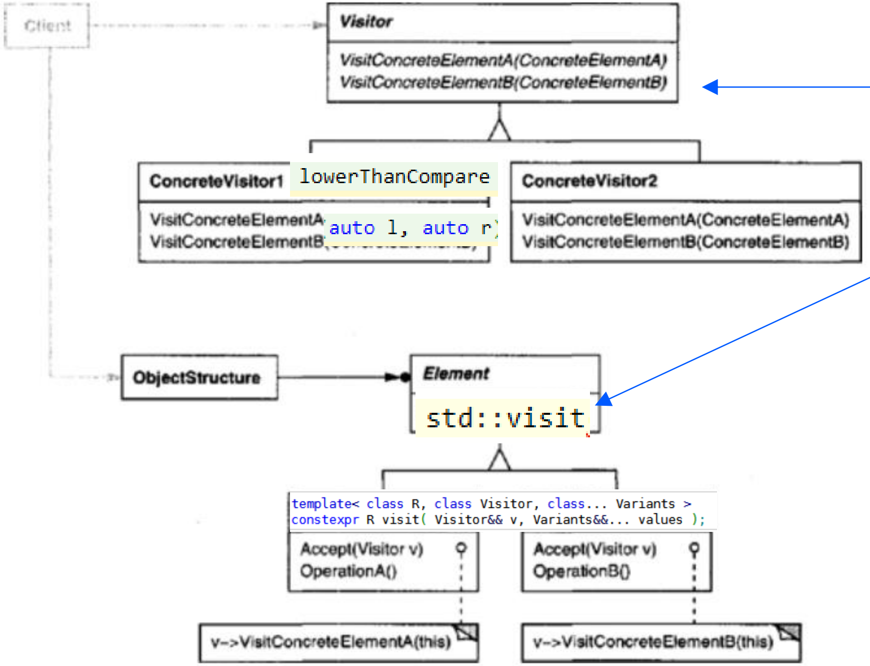


How does this map to each other?

```
auto lowerThanCompare = [](auto l, auto r) {
    return l < r;
};

std::cout << std::visit(lowerThanCompare, douglas, pi) << '\n' ;
```

Visitor

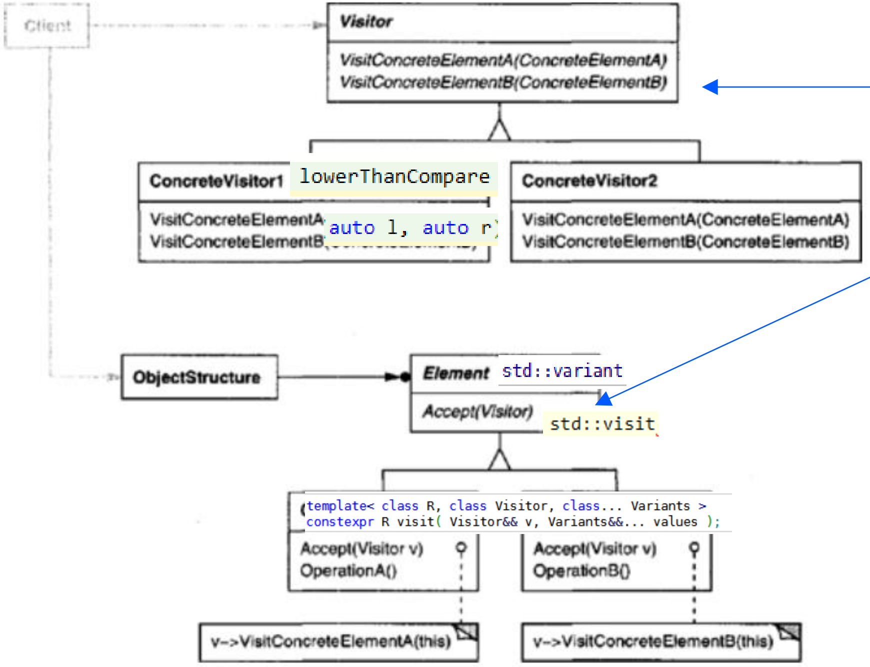


This is generic
Therefore the polymorphism here is achieved using templates/overloads!

BTW, BOOST uses name `accept_visitor`

```
auto lowerThanCompare = [](auto l, auto r) {
    return l < r;
};
std::cout << std::visit(lowerThanCompare,
douglas, pi) << '\n' ;
```


Visitor



This is generic
Therefore the polymorphism here is achieved using templates/overloads!

BTW, BOOST uses name `accept_visitor`

```
auto lowerThanCompare = [](auto l, auto r) {
    return l < r;
};
std::cout << std::visit(lowerThanCompare, douglas, pi) << '\n' ;
```


Variant

```
std::variant<std::vector<std::byte>, int> receiveOneKb()
```

```
auto expectErrno = [](auto x) {
    if constexpr(std::is_same_v<int, decltype(x)>) {
        EXPECT_EQ(x, EWOULDBLOCK);
    } else   if constexpr(std::is_same_v<std::vector<std::byte>, decltype(x)>) {
        FAIL();
    }
};
std::visit(expectErrno, client.receiveOneKb());
```

Variant

```
std::variant<std::vector<std::byte>, int> receiveOneKb()
```

```
auto expectErrno = [](auto x) {
    if constexpr(std::is_same_v<int, decltype(x)>) {
        EXPECT_EQ(x, EWOULDBLOCK);
    } else  if constexpr(std::is_same_v<std::vector<std::byte>, decltype(x)>) {
        FAIL();
    }
};
std::visit(expectErrno, client.receiveOneKb());
```

BTW, VariantWith works as good and is cleaner

Variant

```
std::variant<std::vector<std::byte>, int> receiveOneKb()
```

```
EXPECT_THAT(client.receiveOneKb(),
    VariantWith<std::vector<std::byte>>(expectedTestData));
```

https://godbolt.org/z/1Wvq1d9GE

Function objects and lambdas

```
auto printInt1 = [](int x) { std::cout << x << '\n'; };
printInt1(3);
```

Function objects and lambdas

```cpp
auto printInt1 = [](int x) { std::cout << x << '\n'; };
printInt1(3);
```

```cpp
struct IntPrinter
{
    IntPrinter() = default;
    auto operator()(int x) const {std::cout << x << '\n';}
};

IntPrinter printInt2;
printInt2(42);
```

https://godbolt.org/z/4ffz49edW

Function objects and lambdas

```
auto printInt1 = [](auto x) { std::cout << x << '\n'; };
printInt1(3);
```

```
???????????
```

```
IntPrinter printInt2;
printInt2(42);
```

Function objects and lambdas

```cpp
auto printInt1 = [](auto x) { std::cout << x << '\n'; };
printInt1(3);
```

```cpp
struct IntPrinter
{
    IntPrinter() = default;
    template<typename T>
    auto operator()(T x) const {std::cout << x << '\n';}
};
IntPrinter printInt2;
printInt2(42);
```

Notice that they are templates!

https://godbolt.org/z/e7nnfErhq

Function objects and lambdas

```cpp
int x = 3;
auto printInt1 = [mx = x]() { std::cout << mx << '\n'; };
printInt1();
```

```cpp
struct IntPrinter
{
    IntPrinter(int x) : mx(x) {}
    auto operator()() const {std::cout << mx << '\n';}
    int mx;
};
```

```cpp
int y = 42;
IntPrinter printInt2{y};
printInt2();
```

https://godbolt.org/z/zbEd1qhcq

Function objects and lambdas

```
int x = 3;
auto printInt1 = [&mx = x]() { std::cout << ++mx << '\n'; };
printInt1();
```

https://godbolt.org/z/94PsfY6G3

```
struct IntPrinter
{
    IntPrinter(int& x) : mx(x) {}
    auto operator()() const {std::cout << ++mx << '\n';}
    int& mx;
};

int y = 42;
IntPrinter printInt2{y};
printInt2();
```

Function objects and lambdas

```cpp
struct IntPrinter
{
    IntPrinter(const int& x) : mx(x) {}
    auto operator()() const {std::cout << mx << '\n';}
    const int& mx;
};

int y = 42;
IntPrinter printInt2{y};
printInt2();
```


Function objects and lambdas

```
int x = 3;
auto printInt1 = [&mx = std::as_const(x)]()
{ std::cout << mx << '\n'; };
printInt1();
```

https://godbolt.org/z/Y1YP3KrvE

```
struct IntPrinter
{
    IntPrinter(const int& x) : mx(x) {}
    auto operator()() const {std::cout << mx << '\n';}
    const int& mx;
};

int y = 42;
IntPrinter printInt2{y};
printInt2();
```

Function objects and lambdas

```
auto counter1 = [x=3]() { return x++; };
std::cout << counter1() << ' ' << counter1() << '\n';
```

```
<source>:13:38: error: increment of read-only variable 'x'
   13 |     auto counter1 = [x=3]() { return x++; };
```

https://godbolt.org/z/aea6hMz8d

Function objects and lambdas

```
auto counter1 = [x=3]() { return x++; };
std::cout << counter1() << ' ' << counter1() << '\n';
```

```
struct IntPrinter
{
    IntPrinter(const int& x) : mx(x) {}
    auto operator()() const {std::cout << mx << '\n';}
    int mx;
};
```

```
<source>:13:38: error: increment of read-only variable 'x'
   13 |     auto counter1 = [x=3]() { return x++; };
```

https://godbolt.org/z/aea6hMz8d

Function objects and lambdas

```
auto counter1 = [x=3]() mutable { return x++; };
std::cout << counter1() << ' ' << counter1() << '\n';
```

```
struct Counter
{
    Counter(int x_) : x(x_) {}
    auto operator()() {return x++;}
    int x;
};

int y = 42;
Counter counter2{42};
std::cout << counter2() << ' ' << counter2() << '\n';
```

https://godbolt.org/z/eTrnadTGq

Function objects and lambdas

```
auto factorial = [](unsigned x) -> unsigned {
    if (x > 2) return this->operator()(x-1);
    else return 1;
};
```

Function objects and lambdas

```
auto factorial = [](unsigned x) -> unsigned {
    if (x > 2) return this->operator()(x-1);
    else return 1;
};
```

```
<source>:16:27: error: 'this' was not captured for this lambda function
   16 |         if (x > 2) return this->operator()(x-1);
      |                           ^~~~
Compiler returned: 1
```

https://godbolt.org/z/17sP1vaoo

Function objects and lambdas

```
auto factorial = [this](unsigned x) -> unsigned {
    if (x > 2) return this->operator()(x-1);
    else return 1;
};
```

```
<source>: In function 'int main()':
<source>:15:23: error: invalid use of 'this' in non-member function
   15 |     auto factorial = [this](unsigned x) -> unsigned {
      |                       ^~~~
<source>: In lambda function:
<source>:16:27: error: 'this' was not captured for this lambda function
   16 |         if (x > 2) return this->operator()(x-1);
      |                           ^~~~
Compiler returned: 1
```

https://godbolt.org/z/e3G7zMEeY

Function objects and lambdas

```
auto factorial = [&factorial](unsigned x) -> unsigned {
    if (x > 2) return factorial()(x-1);
    else return 1;
};
```

```
<source>: In function 'int main()':
<source>:15:24: error: use of 'factorial' before deduction of 'auto'
   15 |     auto factorial = [&factorial](unsigned x) -> unsigned {
      |                        ^~~~~~~~~
<source>: In lambda function:
<source>:16:27: error: 'this' was not captured for this lambda function
   16 |         if (x > 2) return this->operator()(x-1);
      |                           ^~~~
Compiler returned: 1
```

Function objects and lambdas

```
auto factorial = [&factorial](unsigned x) -> unsigned {
    if (x > 2) return factorial()(x-1);
    else return 1;
};
```

????????

Function objects and lambdas

```cpp
auto factorial = [](unsigned x, auto&& f) -> unsigned {
    if (x > 1) return x*f(x-1, f);
    else return 1;
};
```


Function objects and lambdas

```
auto factorial = [](unsigned x, auto&& f) -> unsigned {
    if (x > 1) return x*f(x-1, f);
    else return 1;
};
```

```
struct Factorial
{
    Factorial()=default;
    auto operator()(unsigned x) const -> unsigned  {
        if (x > 1) return x*(*this)(x-1);
        else return 1;
    }
};
```

When recursion is involved, use regular object/function for simplicity?

https://godbolt.org/z/En8rxoK1j

Function objects and lambdas

```
std::variant<std::vector<std::byte>,int>
```

This variant can hold one of **two** types

This is a template generic lambda (template).
What exactly will instantiate?

```
auto expectErrno = [](auto x) {
    if constexpr(std::is_same_v<int, decltype(x)>) {
        EXPECT_EQ(x, EWOULDBLOCK);
    } else if constexpr(std::is_same_v<std::vector<std::byte>, decltype(x)>) {
        FAIL();
    }
};
```


Function objects and lambdas

```
std::variant<std::vector<std::byte>,int>
```

This variant can hold one of **two** types

This is a template generic lambda (template).
What exactly will instantiate?

```
auto expectErrno = [](auto x) {
    if constexpr(std::is_same_v<int, decltype(x)>) {
        EXPECT_EQ(x, EWOULDBLOCK);
    } else if constexpr(std::is_same_v<std::vector<std::byte>, decltype(x)>) {
        FAIL();
    }
};
```

https://godbolt.org/z/7643jWK9E

```
::TestBody()::'lambda'(auto)::operator()<int>(auto) const:
```

```
t::TestBody()::'lambda'(auto)::operator()<std::vector<std::byte, std::allocator<std::byte>>>(auto) const:
```


Function objects and lambdas

Let's replace the lambda with a simpler one, only one type is used anyway

```
auto expectErrno = [](auto x) {
    if constexpr(std::is_same_v<int, decltype(x)>) {
        EXPECT_EQ(x, EWOULDBLOCK);
    } else {
        static_assert(dependentFalse<decltype(x)>, "not gonna reach here anyway");
    }
```

```
auto expectErrno = [](int x) {
    EXPECT_EQ(x, EWOULDBLOCK);
};
```

Function objects and lambdas

Let's replace the lambda with a simpler one, only one type is used anyway

```
auto expectErrno = [](auto x) {
    if constexpr(std::is_same_v<int, decltype(x)>) {
        EXPECT_EQ(x, EWOULDBLOCK);
    } else {
        static_assert(dependentFalse<decltype(x)>, "not gonna reach here anyway");
    }
```

```
auto expectErrno = [](int x) {
    EXPECT_EQ(x, EWOULDBLOCK);
};
```

https://godbolt.org/z/xPoz34nM4 https://godbolt.org/z/vd3oM54ah

Function objects and lambdas

Let's replace the lambda with a simpler one, only one type is used anyway

```
auto expectErrno = [](auto x) {
    if constexpr(std::is_same_v<int, decltype(x)>) {
        EXPECT_EQ(x, EWOULDBLOCK);
    } else {
        static_assert(dependentFalse<decltype(x)>, "not gonna reach here anyway");
    }
```

```
auto expectErrno = [](int x) {
    EXPECT_EQ(x, EWOULDBLOCK);
};
```

https://godbolt.org/z/xPoz34nM4 https://godbolt.org/z/vd3oM54ah

Those errors seem cryptic. What they indicate is that visitor has to cover every possible type possible held by variant.

Function objects and lambdas

How else can this be rewritten?

```
auto expectErrno = [](auto x) {
    if constexpr(std::is_same_v<int, decltype(x)>) {
        EXPECT_EQ(x, EWOULDBLOCK);
    } else if constexpr(std::is_same_v<std::vector<std::byte>, decltype(x)>) {
        FAIL();
    }
};
```


Function objects and lambdas

How else can this be rewritten?

```
auto expectErrno = [](auto x) {
    if constexpr(std::is_same_v<int, decltype(x)>) {
        EXPECT_EQ(x, EWOULDBLOCK);
    } else if constexpr(std::is_same_v<std::vector<std::byte>, decltype(x)>) {
        FAIL();
    }
};
```

Using normal function object and function overloading:

```
struct RxFailedResultMatcher
{
    void operator()(int e) const {
        EXPECT_EQ(e, EWOULDBLOCK);
    }
    void operator()(const std::vector<std::byte>&) const {
        FAIL();
    }
};
```

https://godbolt.org/z/xMGGoqY65

Function objects and lambdas

How else can this be rewritten?

```
auto expectErrno = [](auto x) {
    if constexpr(std::is_same_v<int, decltype(x)>) {
        EXPECT_EQ(x, EWOULDBLOCK);
    } else if constexpr(std::is_same_v<std::vector<std::byte>, decltype(x)>) {
        FAIL();
    }
};
```

Using lambda and overload pattern

```
template<typename ...Args>
struct Overload : Args...
{
    using Args::operator()...;
};

auto expectErrno = [](int x) { EXPECT_EQ(x, EWOULDBLOCK); };
auto failOnData = [](const std::vector<std::byte>&) { FAIL(); };
std::visit(Overload{expectErrno, failOnData}, client.receiveOneKb());
```

https://godbolt.org/z/71e4fG8dx

Function objects and lambdas

How else can this be rewritten?

```
auto expectErrno = [](auto x) {
    if constexpr(std::is_same_v<int, decltype(x)>) {
        EXPECT_EQ(x, EWOULDBLOCK);
    } else if constexpr(std::is_same_v<std::vector<std::byte>, decltype(x)>) {
        FAIL();
    }
};
```

Using lambda and overload pattern

```
template<typename ...Args>
struct Overload : Args...
{
    using Args::operator()...;
};

auto expectErrno = [](int x) { EXPECT_EQ(x, EWOULDBLOCK); };
auto failOnData = [](const std::vector<std::byte>&) { FAIL(); };
std::visit(Overload{expectErrno, failOnData}, client.receiveOneKb());
```

https://godbolt.org/z/71e4fG8dx

Note, that this `Overload{expectErrno, failOnData}` uses compile-time class template argument type deduction

Function objects and lambdas

How else can this be rewritten?

```
auto expectErrno = [](auto x) {
    if constexpr(std::is_same_v<int, decltype(x)>) {
        EXPECT_EQ(x, EWOULDBLOCK);
    } else if constexpr(std::is_same_v<std::vector<std::byte>, decltype(x)>) {
        FAIL();
    }
};
```

Using lambda and overload pattern

```
template<typename ...Args>
struct Overload : Args...
{
    using Args::operator()...;
};
template<typename...Args>
Overload(Args...) -> Overload<Args...>;
auto expectErrno = [](int x) { EXPECT_EQ(x, EWOULDBLOCK); };
auto failOnData = [](const std::vector<std::byte>&) { FAIL(); };
std::visit(Overload{expectErrno, failOnData}, client.receiveOneKb());
```

https://godbolt.org/z/5GfaqEz6x

C++17 requires additional deduction guide

Function objects and lambdas

Partial function application and argument binding

```
auto addOne = [](int x) { return x + 1;};
std::cout << addOne(5) << '\n';
```

https://godbolt.org/z/73r5eno6E

But what about this:

```
class Integer
{
private:
    int val;
public:
    Integer(int x) : val(x) {}
    void add(const Integer& that) { val+=that.val; }
    friend std::ostream& operator<<(std::ostream& os, const Integer& i) {
return os << i.val; }
};
```


Function objects and lambdas

Partial function application and argument binding

```
class Integer
{
private:
    int val;
public:
    Integer(int x) : val(x) {}
    void add(const Integer& that) { val+=that.val; }
    friend std::ostream& operator<<(std::ostream& os, const Integer& i) {
return os << i.val; }
};
```


Function objects and lambdas

Partial function application and argument binding

```
class Integer
{
private:
    int val;
public:
    Integer(int x) : val(x) {}
    void add(const Integer& that) { val+=that.val; }
    friend std::ostream& operator<<(std::ostream& os, const Integer& i) {
return os << i.val; }
};

 Integer i{3};
 auto addToI = [&i](const Integer& x) { i.add(x); };
 addToI(Integer{5});
```

https://godbolt.org/z/dqaTMjEPa

Function objects and lambdas

Partial function application and argument binding

```
class Integer
{
private:
    int val;
public:
    Integer(int x) : val(x) {}
    void add(const Integer& that) { val+=that.val; }
    friend std::ostream& operator<<(std::ostream& os, const Integer& i) {
return os << i.val; }
};

Integer i{3};
auto addOneToInteger = [](Integer& x) { x.add(Integer{1}); };
addOneToInteger(i);
std::cout << i << '\n';
```

https://godbolt.org/z/z3GGj5zos

Function objects and lambdas

Partial function application and argument binding

```
class Integer
{
private:
    int val;
public:
    Integer(int x) : val(x) {}
    void add(const Integer& that) { val+=that.val; }
    friend std::ostream& operator<<(std::ostream& os, const Integer& i) {
return os << i.val; }
};

Integer i{3};
auto addOneToInteger = [](Integer& x) { x.add(Integer{1}); };
addOneToInteger(i);
std::cout << i << '\n';
```

https://godbolt.org/z/z3GGj5zos

BTW, compare this against std::bind:

https://en.cppreference.com/w/cpp/utility/functional/bind

Function objects and lambdas

Partial function application and argument binding

```
class Integer
{
private:
    int val;
public:
    Integer(int x) : val(x) {}
    void add(const Integer& that) { val+=that.val; }
    friend std::ostream& operator<<(std::ostream& os, const Integer& i) {
return os << i.val; }
};

Integer i{3};
auto addOneToInteger = [](Integer& x) { x.add(Integer{1}); };
addOneToInteger(i);
std::cout << i << '\n';
```

https://godbolt.org/z/z3GGj5zos

BTW, compare this against std::bind:

https://en.cppreference.com/w/cpp/utility/functional/bind

Function objects and lambdas

Captureless lambdas are convertible to function pointers and **accessed via pointer** act as global functions

```
class Integer
{
private:
    int val;
public:
    Integer(int x) : val(x) {}
    void add(const Integer& that) { val+=that.val; }
    friend std::ostream& operator<<(std::ostream& os, const Integer& i) { return os << i.val; }
};

Integer i{3};
auto addOneToInteger = [](Integer& x) { x.add(Integer{1}); };
// auto addOneToInteger = [one=Integer{1}](Integer& x) { x.add(one); }; // not gonna work
void (*addOneToIntegerPtr)(Integer&) = +addOneToInteger;
addOneToIntegerPtr(i);
```

https://godbolt.org/z/zKKE9hfEv

Function objects and lambdas

Both types are fine to be passed via std::function and similar

```
class Integer
{
private:
    int val;
public:
    Integer(int x) : val(x) {}
    void add(const Integer& that) { val+=that.val; }
    friend std::ostream& operator<<(std::ostream& os, const Integer& i) { return os << i.val; }
};
Integer i{3};
auto addOneToIntegerCaptureless = [](Integer& x) { x.add(Integer{1}); };
auto addOneToIntegerCapturing = [one=Integer{1}](Integer& x) { x.add(one); };
std::function<void(Integer&)> addOne1 = addOneToIntegerCaptureless;
std::function<void(Integer&)> addOne2 = addOneToIntegerCapturing;
addOneToIntegerCaptureless(i);
addOneToIntegerCapturing(i);
```

https://godbolt.org/z/dW93nohr8

Function objects and lambdas

Captureless lambdas are convertible to function pointers and **accessed via pointer** act as global functions

```
class Integer
{
private:
    int val;
public:
    Integer(int x) : val(x) {}
    void add(const Integer& that) { val+=that.val; }
    friend std::ostream& operator<<(std::ostream& os, const Integer& i) { return os << i.val; }
};

Integer i{3};
auto addOneToInteger = [](Integer& x) { x.add(Integer{1}); };
// auto addOneToInteger = [one=Integer{1}](Integer& x) { x.add(one); }; // not gonna work
void (*addOneToIntegerPtr)(Integer&) = +addOneToInteger;
addOneToIntegerPtr(i);
```

Lambdas have distinct types inside translation unit. DO NOT PASS THEM AROUND DIRECTLY WITH THEIR TYPE KNOWN, AS THIS MIGHT LEAD TO ODR VIOLATIONS

https://godbolt.org/z/zKKE9hfEv

Function objects and lambdas

```
auto createCounter()
{
    return [i=unsigned{0u}]  () mutable { return i++; };
}
```

https://godbolt.org/z/Kc3hj45vE

This compiles. If used inside a single cpp file, it is OK.
If passing the return value between files, erase the type of the lambda, e.g.

```
auto createCounter() -> std::function<unsigned()>
{
    return [i=unsigned{0u}]  () mutable { return i++; };
}
```

https://godbolt.org/z/K4PnM3xKb

https://godbolt.org/z/zKKE9hfEv

Function objects and lambdas

Per analogiam

```
auto createOneAdder()
{
    return [] (int x)  { return x+1; };
}
```

https://godbolt.org/z/aKEfeMrEn

Prefer not to, unless in a single translation unit

```
auto createCounter() -> std::function<int(int)>
{
    return [i=unsigned{0u}]  () mutable { return i++; };
}
```

https://godbolt.org/z/PK6r3a9K4

This is safe

```
int (* createOneAdder()) (int)
{
    return [] (int x)  { return x+1; };
}
```

https://godbolt.org/z/hK9T4hWez

And so is this (de facto global function, remeber?)

Function objects and lambdas

Per analogiam

```
auto createOneAdder()
{
    return [] (int x)  { return x+1; };
}
```

https://godbolt.org/z/aKEfeMrEn

Prefer not to, unless in a single translation unit

```
auto createCounter() -> std::function<int(int)>
{
    return [i=unsigned{0u}]  () mutable { return i++; };
}
```

https://godbolt.org/z/PK6r3a9K4

This is safe

```
int (* createOneAdder()) (int) // OMG HOW UGLY THIS IS ☹
{
    return [] (int x)  { return x+1; };
}
```

https://godbolt.org/z/hK9T4hWez

And so is this (de facto global function, remeber?)

Function objects and lambdas

Per analogiam

```
auto createOneAdder()
{
    return [] (int x)  { return x+1; };
}
```

https://godbolt.org/z/aKEfeMrEn

Prefer not to, unless in a single translation unit

```
auto createCounter() -> std::function<int(int)>
{
    return [i=unsigned{0u}]  () mutable { return i++; };
}
```

https://godbolt.org/z/PK6r3a9K4

This is safe

```
auto createOneAdder() -> int(*)(int)
{
    return [] (int x)  { return x+1; };
}
```

https://godbolt.org/z/hK9T4hWez

And so is this (de facto global function, remeber?)

Function objects and lambdas

Per analogiam

```
auto createOneAdder()
{
    return [] (int x)  { return x+1; };
}
```

https://godbolt.org/z/aKEfeMrEn

Prefer not to, unless in a single translation unit

```
auto createCounter() -> std::function<int(int)>
{
    return [i=unsigned{0u}]  () mutable { return i++; };
}
```

https://godbolt.org/z/PK6r3a9K4

This is safe

```
auto createOneAdder() -> int(*)(int) //much nicer, isn't it?
{
    return [] (int x)  { return x+1; };
}
```

https://godbolt.org/z/GoPnf84Y4

And so is this (de facto global function, remeber?)

Function objects and lambdas

Per analogiam

```
auto createOneAdder()
{
    return [] (int x)  { return x+1; };
}
```

https://godbolt.org/z/aKEfeMrEn

Prefer not to, unless in a single translation unit

```
auto createCounter() -> std::function<int(int)>
{
    return [i=unsigned{0u}]  () mutable { return i++; };
}
```

https://godbolt.org/z/PK6r3a9K4

This is safe

```
auto createOneAdder()
{
    return +[] (int x)  { return x+1; };
}
```

https://godbolt.org/z/5dM1xEcTK

And so is this (de facto global function, remeber?)

Optional

Consider the original socket example, but assume the error code is irrelevant.
What should the interface look like?

```
/* ???? */                    receiveOneKb() noexcept
```


Optional

Consider the original socket example, but assume the error code is irrelevant.
What should the interface look like?

```
std::vector<std::byte> receiveOneKb() noexcept
```

+ Really simple
+ Empty vector indicates error
- Empty RX buffer is a valid situation (well, closed connection, but that’s not an error)

Optional

Consider the original socket example, but assume the error code is irrelevant.
What should the interface look like?

```
std::vector<std::byte>* receiveOneKb() noexcept
std::unique_ptr<std::vector<std::byte>> receiveOneKb() noexcept
```

+ Well defined empty value
- Involves heap allocation

Optional

Consider the original socket example, but assume the error code is irrelevant.
What should the interface look like?

```
std::variant<std::monostate, std::vector<std::byte>> receiveOneKb() noexcept
```

+ Safe
- A bit verbose

Optional

Consider the original socket example, but assume the error code is irrelevant.
What should the interface look like?

```
std::optional<std::vector<std::byte>> receiveOneKb() noexcept
```


Optional

Consider the original socket example, but assume the error code is irrelevant.
What should the interface look like?

```
std::optional<std::vector<std::byte>> receiveOneKb() noexcept
```

+ Stack-allocated
+ Pointer-like semantics
+ Possible safe access
- Unsafe access also possible ☹
- Cannot store references (easily)

```
auto result = client.receiveOneKb();
EXPECT_FALSE(result.has_value());
EXPECT_FALSE(static_cast<bool>(result));
EXPECT_EQ(result, std::nullopt);
EXPECT_THROW(result.value(), std::bad_optional_access);
```

https://godbolt.org/z/h6Ev3o7or

Optional

Consider the original socket example, but assume the error code is irrelevant.
What should the interface look like?

```
std::optional<std::vector<std::byte>> receiveOneKb() noexcept
```

+ Stack-allocated
+ Pointer-like semantics
+ Possible safe access
- Unsafe access also possible ☹
- Cannot store references (easily)

```
auto result = client.receiveOneKb();
EXPECT_FALSE(result.has_value());
EXPECT_FALSE(static_cast<bool>(result));
EXPECT_EQ(result, std::nullopt);
EXPECT_THROW(result.value(), std::bad_optional_access);

(void) *result; //oops, that's UB

return result->size(); //and so is this
```

https://godbolt.org/z/WMP4bTqf9

Optional

Consider the original socket example, but assume the error code is irrelevant.
What should the interface look like?

```
std::variant<std::monostate, std::vector<std::byte>> receiveOneKb() noexcept
```

The above construct can be sometimes considered safer due to lack of (easily accessible) dereferencing operator

Optional

```
struct PhysicalHandle
{
//whatever
};

class Device
{
private:
    std::optional<PhysicalHandle> handle;
public:
    Device() = default;
    std::optional<const PhysicalHandle&> getHandle() const;
};
```

This won’t compile, how to fix it?

https://godbolt.org/z/91s47dY15

Optional

```
struct PhysicalHandle
{
//whatever
};

class Device
{
private:
    std::optional<PhysicalHandle> handle;
public:
    Device() = default;
    const PhysicalHandle* getHandle() const;
};
```

Works, but we lose the safe access possibilites

https://godbolt.org/z/Geh4nEsWq

Optional

```
struct PhysicalHandle
{
//whatever
};

class Device
{
private:
    std::optional<PhysicalHandle> handle;
public:
    Device() = default;
    std::optional<std::reference_wrapper<const PhysicalHandle>> getHandle() const
};
```

OK, but verbose

https://godbolt.org/z/svYhfY8zo

Optional

```
struct PhysicalHandle
{
//whatever
};

class Device
{
private:
    std::optional<PhysicalHandle> handle;
public:
    Device() = default;
    boost::optional<const PhysicalHandle&> getHandle() const
};
```

Works, but pulls in extra dependency

https://godbolt.org/z/9PEohze9a

Optional

```
struct PhysicalHandle
{
//whatever
};

class Device
{
private:
    std::optional<PhysicalHandle> handle;
public:
    Device() = default;
    boost::optional<const PhysicalHandle&> getHandle() const
};
```

Works, but pulls in extra dependency

https://godbolt.org/z/9PEohze9a

Optional

Assignment

```
std::optional<std::vector<std::byte>> dummyData{{std::byte{2}, std::byte{4}}};
dummyData = {std::byte{4}, std::byte{2}};
std::cout <<  static_cast<int>(dummyData.value()[0]) << '\n';
```

```
std::optional<std::vector<std::byte>> dummyData{{std::byte{2}, std::byte{4}}};
*dummyData = {std::byte{4}, std::byte{2}};
std::cout <<  static_cast<int>(dummyData.value()[0]) << '\n';
```

BAD, works by sheer luck

Optional

Assignment

```
std::optional<std::vector<std::byte>> dummyData{{std::byte{2}, std::byte{4}}};
dummyData = {std::byte{4}, std::byte{2}};
std::cout <<  static_cast<int>(dummyData.value()[0]) << '\n';
```

```
std::optional<std::vector<std::byte>> dummyData{{std::byte{2}, std::byte{4}}};
*dummyData = {std::byte{4}, std::byte{2}};
std::cout <<  static_cast<int>(dummyData.value()[0]) << '\n';
```

BAD, works by sheer luck

```
std::optional<std::vector<std::byte>> dummyData{};
*dummyData = {std::byte{4}, std::byte{2}};
std::cout << dummyData.has_value();
```

UB galore!!!

Optional

Empty -> non-empty

Simply assign:

```
std::optional<std::vector<std::byte>> dummyData{{std::byte{2}, std::byte{4}}};
dummyData = {std::byte{4}, std::byte{2}};
std::cout <<  static_cast<int>(dummyData.value().front()) << '\n';
```

...or use emplace (and utilize the fact it return a reference to newly created object)

```
std::optional<std::vector<std::byte>> dummyData{{std::byte{2}, std::byte{4}}};
std::cout <<  static_cast<int>(dummyData.emplace({std::byte{4}, std::byte{2}}).front()) << '\n';
```

https://godbolt.org/z/Tv7nsMhqz

Optional

Non-empty -> empty

```
std::optional<int> oi{32};
std::cout << oi.has_value() << '\n';
oi.reset();
std::cout << oi.has_value() << '\n';
```

https://godbolt.org/z/hY5x616cs

Optional

Extraction/dereference

Pointer-style

```
if (oi.has_value()) {
    std::cout << *oi << '\n’; //or -> when possible
} else {
    std::cout << -1 << '\n';
}
```

https://godbolt.org/z/ehKf1vEsv

Throw on failed dereference (possibly to propagate it higher up the callstack)

```
try {
    std::cout << oi.value() << '\n';
} catch(const std::bad_optional_access&) {
    std::cout << -1 << '\n';
}
```

Return a default value: `std::cout << oi.value_or(-1) << '\n';`

Optional

Comparison

Lots of pitfalls.
Simply put: two empty optionals<T>/nullopts are equal.
Two non-empty optionals are equal if values they hold are equal.

Empty optional is less than anything non-empty.

```
std::optional<int> big{32};
std::optional<int> small{2};
std::optional<int> empty{};

std::cout << (empty == empty) << '\n'; //1
std::cout << (empty == small) << '\n'; //0
std::cout << (big == small) << '\n'; //0
std::cout << (small<big) << '\n'; //1
std::cout << (empty<empty) <<'\n'; //0
std::cout << (big<empty) <<'\n'; //0
std::cout << (small<empty) <<'\n'; //0
std::cout << (empty<small) <<'\n'; //1
```

https://godbolt.org/z/aoEb18b97

Optional

Moved-from optional

```
auto src = std::make_optional(std::vector<int>{1,2,3});
std::cout << src.has_value() << '\n';
auto dst = std::move(src);
std::cout << src.has_value() << '\n';
std::cout << dst.has_value() << '\n';
std::cout << src->size() << '\n';
std::cout << dst->size() << '\n';
```

src is being moved from

yet it still remains non-empty optional

dst is properly move-constructed

src's inner object is actually moved from into dst

https://godbolt.org/z/Ke1f8dh5c

NOKIA

Going further

In-place construction

```cpp
#define LOG_FUNC_NAME() \
    do { \
        auto l=std::source_location::current(); \
        std::cout << l.function_name() <<'\n'; \
    } while(0);

struct Logged
{
    Logged() {LOG_FUNC_NAME();}
    Logged(const Logged&) {LOG_FUNC_NAME();}
    Logged(Logged&&) noexcept {LOG_FUNC_NAME();}
    Logged& operator=(const Logged&) noexcept {LOG_FUNC_NAME(); return *this;}
    Logged& operator=(Logged&&) noexcept {LOG_FUNC_NAME(); return *this;}
};
```

```cpp
std::optional<Logged> ol{Logged{}};
std::variant<std::monostate, Logged> lv{Logged{}};
```

What is the logger class going to print?

Going further

In-place construction

```
#define LOG_FUNC_NAME() \
    do { \
        auto l=std::source_location::current(); \
        std::cout << l.function_name() <<'\n'; \
    } while(0);

struct Logged
{
    Logged() {LOG_FUNC_NAME();}
    Logged(const Logged&) {LOG_FUNC_NAME();}
    Logged(Logged&&) noexcept {LOG_FUNC_NAME();}
    Logged& operator=(const Logged&) noexcept {LOG_FUNC_NAME(); return *this;}
    Logged& operator=(Logged&&) noexcept {LOG_FUNC_NAME(); return *this;}
};
```

```
std::optional<Logged> ol{Logged{}};
std::variant<std::monostate, Logged> lv{Logged{}};
```

What is the logged class going to print?

```
Logged::Logged()
Logged::Logged(Logged&&)
Logged::Logged()
Logged::Logged(Logged&&)
```

https://godbolt.org/z/s6PMxe88W

Going further

In-place construction

```
std::optional<Logged> ol{Logged{}};

std::variant<std::monostate, Logged> lv{Logged{}};
```

```
Logged::Logged()
Logged::Logged(Logged&&)
Logged::Logged()
Logged::Logged(Logged&&)
```

Going further

In-place construction

```
std::optional<Logged> ol{std::in_place};
std::variant<std::monostate, Logged> lv{std::in_place_index<1>};
```

https://godbolt.org/z/Y7ojnGvn7

```
Logged::Logged()
Logged::Logged()
```

```
std::optional<Logged> ol{std::in_place};
std::variant<std::monostate, Logged> lv{std::in_place_type<Logged>};
```

https://godbolt.org/z/8o9z911K9

Going further

Runtime polymorphism without virtual functions and heap allocations

```
struct Logger
{
    ~Logger() = default;
    virtual void print(std::string_view) const = 0;
};
struct ErrLogger : Logger
{
    void print(std::string_view s) const override {
        std::cout << "ERR " << s << '\n';
    }
};
struct InfLogger : Logger
{
    void print(std::string_view s) const override {
        std::cout << "INF " << s << '\n';
    }
};
```

```
void logMsg(const Logger& l, std::string_view msg)
{
    l.print(msg);
}

int main()
{
    std::unique_ptr<Logger> infoLogger = std::make_unique<InfLogger>();
    std::unique_ptr<Logger> errorLogger = std::make_unique<ErrLogger>();

    logMsg(*infoLogger, "ala ma kota");
    logMsg(*errorLogger, "ups!");
}
```

https://godbolt.org/z/58PYYjG8z

Going further

Runtime polymorphism without virtual functions and heap allocations

```cpp
struct ErrLogger
{
    void print(std::string_view s) const
    {
        std::cout << "ERR " << s << '\n';
    }
};

struct InfLogger
{
    void print(std::string_view s) const
    {
        std::cout << "INF " << s << '\n';
    }
};
```

```cpp
void logMsg(const std::variant<ErrLogger, InfLogger>& l, std::string_view msg)
{
    std::visit([msg] (const auto& l) {l.print(msg);}, l);
}

int main()
{
    InfLogger infoLogger{};
    ErrLogger errorLogger{};
    logMsg(infoLogger, "ala ma kota");
    logMsg(errorLogger, "ups!");
}
```

https://godbolt.org/z/ejxnxx9xK

Going further

Safer optional

```
std::optional<float> of{std::in_place, 12.3};
std::optional<int> oi;
if (of) {
    oi = static_cast<int>(*of);
}

if (oi) {
    std::cout << *oi <<'\n';
}
```

```
opt::option<float> of{12.3};
of
    .map([](float x) { return static_cast<int>(x); })
    .inspect([](int x){std::cout << x << '\n';});
```

https://godbolt.org/z/1Ev5dqd6s

https://godbolt.org/z/nPG1hadhG

https://github.com/NUCLEAR-BOMB/option

https://github.com/TartanLlama/optional

Going further

More expressive return values

```
std::variant<std::vector<std::byte>, int> receiveOneKb()
```

```
std::expected<std::vector<std::byte>, int> receiveOneKb() noexcept
    {
        static constexpr auto kilobyteBytes=1024u;
        std::vector<std::byte> rxdata(kilobyteBytes);
        auto nreceived = socket->receive(rxdata.data(), kilobyteBytes);
        if (nreceived < 0) {
            return std::unexpected(errno);
        }
        if (nreceived < kilobyteBytes) {
            rxdata.resize(static_cast<std::size_t>(nreceived));
        }
        return rxdata;
    }
};
```

C++23, boost::outcome etc. …

https://godbolt.org/z/rch9v9Yc7

Recommended reading

1. www.cppreference.com
2. Design Patterns: Elements of Reusable Object-Oriented, Gamme E., Helm R., Johnson R., Vlissides J.
3. Modern C++ Design: Generic Programming and Design Patterns, Alexandrescu A., Lafferty D.

Questions?

