Lambdas, how to capture everything and stay sane

Dawid Zalewski

github.com/zaldawid
zaldawid@gmail.com
saxion.edu

11/19/22

Rule #1:

almost always

Capturing is about lifetime.

A casual occurrence

```
auto add_record(record const& rec) {
  auto key{ generate_key() };
  store_update(db_context, key, rec);
  return [=](){ return store_retrieve(db_context, key); };
}
```

Capture-default, everything what's needed is copied into the lambda.

```
auto accessor{ add_record( record{/*~~~*/}) };

std::cout << accessor();
```

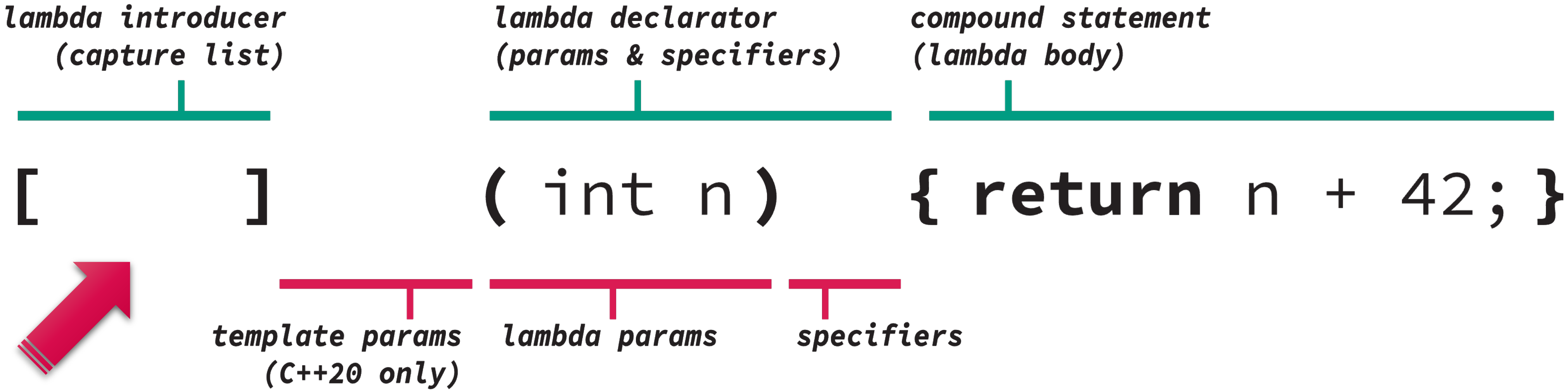
A casual occurrence

```
auto add_record(record const& rec) {
```

```
terminate called after throwing an instance of 'std::bad_alloc'
  what():  std::bad_alloc
Aborted
```

```
std::cout << accessor();
```

Lambdas' anatomy



lambda introducer (capture list)

lambda declarator (params & specifiers)

compound statement (lambda body)

```
[     ]   ( int n )  { return n + 42; }
```

template params (C++20 only)

lambda params

specifiers

Lambdas & closures

Lambda expression

```
auto lambda = [](int n) {
    return n + 42;
  };
```

Closure type

```
class __lambda_class {
public:


  int operator()(int n) const {
    return n + 42;
  };


};
```

Lambdas & closures

Lambda expression

```
void func(){
  auto k{42};

  auto lambda = [](int n) {
      return n + k;
    };
}
```

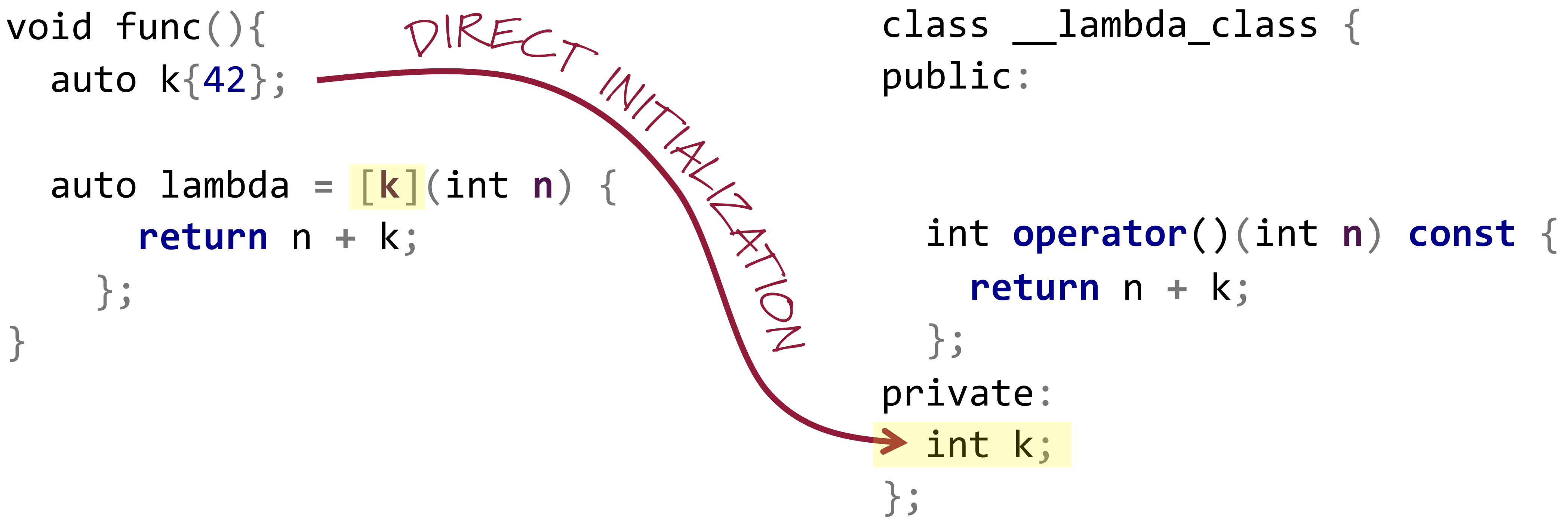
Closure type

```
class __lambda_class {
public:


  int operator()(int n) const {
    return n + k;
  };


};
```

Lambdas & closures



Lambda expression

```
void func(){
  auto k{42};

  auto lambda = [k](int n) {
      return n + k;
    };
}
```

Closure type

```
class __lambda_class {
public:


  int operator()(int n) const {
    return n + k;
  };
private:
  int k;
};
```

DIRECT INITIALIZATION

Simple captures & capture defaults

```
void func(){
  auto n = 42;
  auto k = 11;

  auto l1 = [=]     () { return k + n; };

  auto l2 = [&]     () { return k + n; };

  auto l3 = [n]     () { return k + n; };

  auto l4 = [=, &k] () { return k + n; };

}
```

← capture default: n & k by copy

← capture default: n & k by reference

← capture default: only n by copy

← mixed capture: k by reference
n by copy

Generalized captures a.k.a init captures

```
void func(){
  auto answer = 42;

  auto l1 = [&m=answer] (int a) {
                        return a + m;
                      };

  auto l2 = [m=func()] (int a) {
                        return a + m;
                       };
}
```

capture::operator()

Lambda expression

```cpp
void func(){
  std::string str = "The answer is: ";
  auto k = 43;

  auto l1 = [&str, k] () {
   return str + std::to_string(--k);
  };

}
```

Closure type

```cpp
class __l1_class {
public:
  int operator()() const {
    return str + std::to_string(--k);
  }
private:
  std::string& str;
  int k;
};
```

```
error: assignment of read-only variable 'k'
```

capture::operator()

Lambda expression

```
void func(){
  std::string str = "The answer is: ";
  auto k = 43;

  auto l1 = [&str, k] () {
   return str + std::to_string(--k);
  };

}
```

Closure type

```
class __l1_class {
public:
  int operator()() const {
    return str + std::to_string(--k);
  }
private:
  std::string& str;
  int k;
};
```

illegal mutation

capture::operator() & references

Lambda expression

```
void func(){
  std::string str = "The answer is: ";
  auto k = 43;

  auto l1 = [&str, &k] () {
   return str + std::to_string(--k);
  };

}
```

Closure type

```
class __l1_class {
public:
  int operator()() const {
    return str + std::to_string(--k);
  }
private:
  std::string& str;
  int &k;
};
```

totally legal mutation 👉

const vs. mutable

Lambda expression

```
void func(){
  std::string str = "The answer is: ";
  auto k = 43;

  auto l1 = [&str, k] () mutable {
   return str + std::to_string(--k);
  };

}
```

Closure type

```
class __l1_class {
public:
  int operator()() ~~const~~ {
    return str + std::to_string(--k);
  }
private:
  std::string& str;
  int k;
};
```

Simple captures by copy, type deduction

Lambda expression

```
void func(){
  std::string str = "The answer is: ";
  const auto k = 43;

  auto l1 = [&str, k] () mutable {
   return str + std::to_string(--k);
  };

}
```

Closure type

```
class __l1_class {
public:
  int operator()() {
    return str + std::to_string(--k);
  }
private:
  std::string& str;
  const int k;
};
```

```
error: assignment of read-only variable 'k'
```

Rule #2:

Get familiar with capture type deduction rules.

Capture type deduction rules

Capture	Equivalent syntax	*cv-* qualifiers
`&var`	`auto& var = var`	cv preserved
`&var=init`	`auto& var = init`	cv preserved
`var`	`---`	cv preserved
`var=init`	`auto var = init`	cv dropped

cv- **cannot be added when capturing by copy.**

Simple captures by copy, type deduction

Lambda expression

```
void func(){
  std::string str = "The answer is: ";
  const auto k = 43;

  auto l1 = [&str, k] () mutable {
   return str + std::to_string(--k);
  };

}
```

Closure type

```
class __l1_class {
public:
  int operator()() {
    return str + std::to_string(--k);
  }
private:
  std::string& str;
  const int k;
};
```

illegal mutation

Dropping const with init capture

Lambda expression

```
void func(){
  std::string str = "The answer is: ";
  const auto k = 43;

  auto l1 = [&str, k=k] () mutable {
   return str + std::to_string(--k);
  };

}
```

Closure type

```
class __l1_class {
public:
  int operator()() {
    return str + std::to_string(--k);
  }
private:
  std::string& str;
  int k;
};
```

Adding const to a by-copy capture

Lambda expression

```
void func(){
  std::string str = "The answer is: ";
  auto k = 42;

  auto l1 = [&str, k] () mutable {
   return str + std::to_string(--k);
  };

}
```

Closure type

```
class __l1_class {
public:
  int operator()() {
    return str + std::to_string(k);
  }
private:
  std::string& str;
  const int k;
};
```

Adding const to a by-copy capture

Won’t work*:

- Adding qualifiers to the capture:

```
[&str, const k=k] () mutable {/*~~~*/};
```

- Using a cast and an init-capture:

```
[&str, k=std::as_const(k)] () mutable {/*~~~*/};
```

*this list is probably not exhaustive

Adding const to a by-copy capture

Lambda expression

```
void func(){
  std::string str = "The answer is: ";
  auto k = 42;
  const auto k_copy = k;

  auto l1 = [&str, k_copy] () mutable {
   return str + std::to_string(k_copy);
  };

}
```

Closure type

```
class __l1_class {
public:
  int operator()() {
    return str + std::to_string(k_copy);
  }
private:
  std::string& str;
  const int k_copy;
};
```

Adding const with init capture

Lambda expression

```
void func(){
  std::string str = "The answer is: ";
  const auto k = 43;

  auto l1 = [&str=std::as_const(str),
             k=k] () mutable {
    return str + std::to_string(--k);
  };
}
```

Closure type

```
class __l1_class {
public:
  int operator()() {
    return str + std::to_string(--k);
  }
private:
  std::string const& str;
  int k;
};
```

Rule #3:

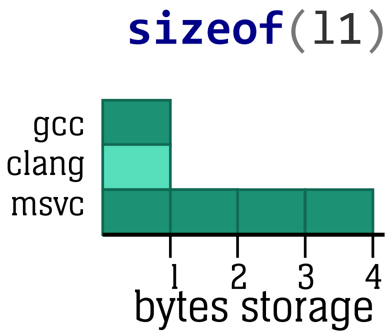
Understand when not to capture.

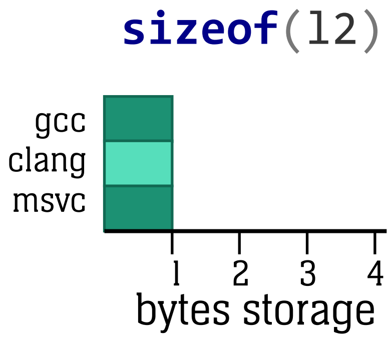
When (not) to capture

```
int square(int num){
   return num * num;
}

int main(){
   const auto answer {42};
   auto l1 = [ = ](){ return answer + square(answer); }; ✓ (capture is not needed)
   auto l2 = [   ](){ return answer + square(answer); }; ✓
}
```

When (not) to capture

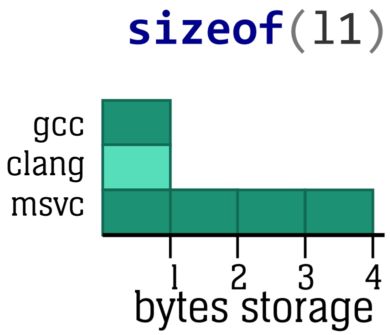


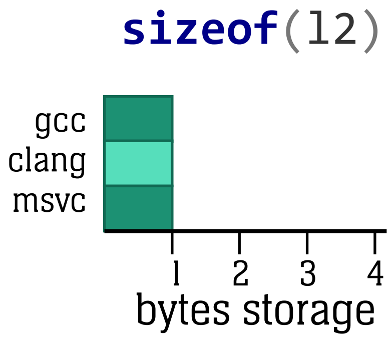


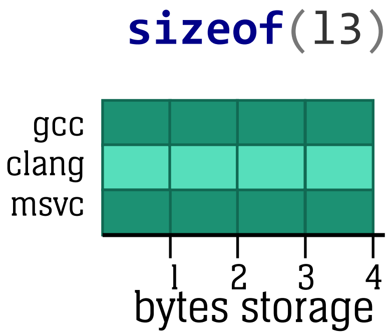
```
int square(int num){
   return num * num;
}

int main(){
   const auto answer {42};
   auto l1 = [ = ](){ return answer + square(answer); }; ✓
   auto l2 = [   ](){ return answer + square(answer); }; ✓
}
```

When (not) to capture







```
int square(int num){
   return num * num;
}


int main(){
   const auto answer {42};
   auto l1 = [ = ](){ return answer + square(answer); };
   auto l2 = [   ](){ return answer + square(answer); };
   auto l3 = [ answer ](){ return answer + square(answer); };
}
```

When ~~(not)~~ to capture

```
int square(int const& num){
   return num * num;
}

int main(){
   const auto answer {42};
   auto l1 = [ = ](){ return answer + square(answer); };
   auto l2 = [   ](){ return answer + square(answer); };      ✗
   auto l3 = [ answer ](){ return answer + square(answer); };
}
```

Address of answer is read when calling func

When probably (not) to capture

```
constexpr      string_view sv{"Hello"};

auto l1 = [=](){ return std::string{sv} + " C++"; };

auto l2 = [] (){ return std::string{sv} + " C++"; };
```

←Works across the board

←Works only with msvc

Rule #4:

You never capture objects with static storage duration.

Static lifetime and captures

```
auto kv_proxy::add_record(record const& rec) {
  auto key{ generate_key() };
  store_update(db_context, key, rec);
  return [=](){ return store_retrieve(db_context, key); };
}

kv_proxy proxy{};

auto accessor = proxy.add_record( record{"Hello, lambdas!"} );

std::cout << accessor();
```

Capture-default, everything what's needed is copied into the lambda.

Static lifetime and captures

```
connection_context db_context{/*...*/};

auto kv_proxy::add_record(record const& rec) {
  auto key{ generate_key() };
  store_update(db_context, key, rec);
  return [=](){ return store_retrieve(db_context, key); };
}
```

db_context has static storage duration

- What is db_context?
- What is copied into the lambda?

Static lifetime == no captures

```
connection_context db_context{/*...*/};

auto kv_proxy::add_record(record const& rec) {
  auto key{ generate_key() };
  store_update(db_context, key, rec);
  return [key, db_context](){ return store_retrieve(db_context, key); };
}
```

db_context has static storage duration

- warning: capture of variable 'db_context' with non-automatic storage duration
- error: 'db_context' cannot be captured because it does not have automatic storage duration
- error C3495: 'db_context': a simple capture must be a variable with automatic storage duration in the reaching scope of the lambda

Static lifetime == no captures

```
connection_context db_context{/*...*/};

auto kv_proxy::add_record(record const& rec) {
  auto key{ generate_key() };
  store_update(db_context, key, rec);
  return [key](){ return store_retrieve(db_context, key); };
}

kv_proxy proxy{};

auto accessor = proxy.add_record( record{"Hello, lambdas!"} );
// on another thread: db_context.reset();
std::cout << accessor(); // 😱 runtime error!
```

db_context has static storage duration

Rule #5:

Copy objects with static lifetime using an init-capture if you depend on their state.

Init captures save the day

```
connection_context db_context{/*...*/};

auto kv_proxy::add_record(record const& rec) {
  auto key{ generate_key() };
  store_update(db_context, key, rec);
  return [key, db_context=db_context](){ return store_retrieve(db_context, key); };
}
```

db_context has static storage duration

How we came here

```
auto kv_proxy::add_record(record const& rec) {
  auto key{ generate_key() };
  store_update(db_context, key, rec);
  return [=](){ return store_retrieve(db_context, key); };
}
```

Capture-default, everything what's needed is copied into the lambda.

Rule #6:

Do not capture everything.

(Never use capture defaults.)

Do not capture everything

```
struct kv_proxy {

  database_context db_context{/*~~~*/};

  auto add_record(record const& rec) {
    auto key{ generate_key() };
    store_update(db_context, key, rec);
    return [=](){ return store_retrieve(db_context, key); };
  }

};
```

Q: What is captured here?

a) key
b) kv_proxy
c) db_context
d) some combination of above

Do not capture everything

```
struct kv_proxy {

  database_context db_context{/*~~~*/};

  auto add_record(record const& rec) {
    auto key{ generate_key() };
    store_update(db_context, key, rec);
    return [key, this](){ return store_retrieve(db_context, key); };
  }

};
```

this captures a reference to
the enclosing kv_proxy object

Do not capture everything

```
auto group_and_add(std::string_view group_name){
  record rec{ record_from_group(group_name) };
  kv_proxy proxy{/*~~~*/};          // create a local proxy object
  return proxy.add_record(rec);     // return a lambda that refers to it
}

auto accessor = group_and_add( "capturing" );


std::cout << accessor(); // 😱 runtime error!
```

the local proxy object is gone after this line

Capturing this, by reference

```
struct kv_proxy {

  database_context db_context{/*~~~*/};

  auto add_record(record const& rec) {
    auto key{ generate_key() };
    store_update(db_context, key, rec);
    return [key, this](){ return store_retrieve(db_context, key); };
  }

};
```

this captures a reference to the enclosing kv_proxy object

Capturing this, by copy

```
struct kv_proxy {

  database_context db_context{/*~~~*/};

  auto add_record(record const& rec) {
    auto key{ generate_key() };
    store_update(db_context, key, rec);
    return [key, *this](){ return store_retrieve(db_context, key); };
  }

};
```

*this captures a copy of the enclosing kv_proxy object

Capturing this

```
return [key,  this](){ return store_retrieve(db_context, key); };

return [key, *this](){ return store_retrieve(db_context, key); };
```

Capturing this

```
return [key,  this](){ return store_retrieve(this->db_context, key); };

return [key, *this](){ return store_retrieve(this->db_context, key); };
```

Rule #7:

Approach capturing this with caution.

The magic of this

Lambda within a class

```
struct A{
  int n;

  auto make_lambda(){
    return [this](){
      n += 42;
      return n;
    };
  }
};
```

Another lambda within a class

```
struct A{
  int n;

  auto make_lambda(){
    return [*this](){
      n += 42;
      return n;
    };
  }
};
```

The magic of this

Lambda within a class

```
struct A{
  int n;

  auto make_lambda(){
    return [this](){
      this->n += 42;
      return this->n;
    };
  }
};
```

Another lambda within a class

```
struct A{
  int n;

  auto make_lambda(){
    return [*this](){
      this->n += 42;
      return this->n;
    };
  }
};
```

The magic of this

Lambda within a class

```
struct A{
  int n;

  auto make_lambda(){
    return [this](){
      this->n += 42;
      return this->n;
    };
  }
};
```

✓ Compiles

Another lambda within a class

```
struct A{
  int n;

  auto make_lambda(){
    return [*this](){
      this->n += 42;
      return this->n;
    };
  }
};
```

✗ Compile does not

The magic of this

Lambda within a class

```
struct A{
  int n;

  auto make_lambda(){
    return [this](){
      this->n += 42;
      return this->n;
    };
  }
};
```

Its closure type

```
class __lambda_class{
public:


  int operator()() const {
    ref_to_this.n += 42;
    return ref_to_this.n;
  }

private:
  A& ref_to_this;
};
```

👈mutation

The magic of this

Another lambda within a class

```
struct A{
  int n;

  auto make_lambda(){
    return [*this]() mutable {
      this->n += 42;
      return this->n;
    };
  }
};
```

Its closure type

```
class __lambda_class{
public:


  int operator()() ~~const~~ {
    copy_of_this.n += 42;  👉
    return copy_of_this.n;
  }

private:
  A copy_of_this;
};
```

~~illegal~~
mutation

The magic of this

Lambda within a class

```
struct A{
  int n;

  auto make_lambda(){
    return [this](){
      this->n += 42;
      return this->n;
    };
  }
};
```

Another lambda within a class

```
struct A{
  int n;

  auto make_lambda(){
    return [*this]() mutable {
      this->n += 42;
      return this->n;
    };
  }
};
```

The magic of this

Lambda within a class

```
struct A{
  int n;
```

```
  }
};
```

Another lambda within a class

```
struct A{
  int n;
```

```
  }
};
```

this has different semantics within a lambda's body depending on the capture.

Capturing this

	Capture	Automatic Variables	Enclosing Object (*this)	How it's normally expressed	How to Refer to Enclosing Object
	&	by reference	by reference		this
	=	by copy	by reference	That's a trap!*	this
	this	---	by reference	&*this	this
C++17	*this	---	by copy	*this	this

* - deprecated in C++20

Rule #8:

Use init-captures for capturing this.

Being explicit with this

Capturing this by reference

```
struct A{
  int n;


  auto make_lambda(){
    return [&ref_A=*this](){
      ref_A.n += 42;
      return ref_A.n;
    };
  }
};
```

Capturing this by copy

```
struct A{
  int n;


  auto make_lambda(){
    return [copy_A=*this]() mutable {
      copy_A.n += 42;
      return copy_A.n;
    };
  }
};
```

Being explicit with this

Capturing this by reference

```
struct A{
  int n;
  char a_lot_of_data[1'024];

  auto make_lambda(){
    return [&ref_A=*this](){
      ref_A.n += 42;
      return ref_A.n;
    };
  }
};
```

Capturing this by copy

```
struct A{
  int n;
  char a_lot_of_data[1'024];

  auto make_lambda(){
    return [copy_A=*this]() mutable {
      copy_A.n += 42;
      return copy_A.n;
    };
  }
};
```

Rule #9:

Consider cherry-picking member variables.

Being explicit with this

Capturing this by reference

```
struct A{
  int n;
  char a_lot_of_data[1'024];

  auto make_lambda(){
    return [&ref_A=*this](){
      ref_A.n += 42;
      return ref_A.n;
    };
  }
};
```

Cherry-picking by copy

```
struct A{
  int n;
  char a_lot_of_data[1'024];

  auto make_lambda() {
    return [copy_of_n=n]() mutable {
      copy_of_n += 42;
      return copy_of_n;
    };
  }
};
```

Being explicit with this

```
class A{
  int n_{0};

public:
  int number() const { return n_; }
  void increment (int inc) { n_ += inc; }

  auto make_lambda() {
    return [this]() {
      this->increment_by(42);
      return this->number();
    };
  }
};
```

Being explicit with this

```
class A{
  int n_{0};

public:
  int number() const { return n_; }
  void increment (int inc) { n_ += inc; }

  auto make_lambda() const {
    return [*this]() mutable {
      this->increment_by(42);
      return this->number();
    };
  }
};
```

Being explicit with this

```cpp
class A{
  int n_{0};

public:
  int number() const { return n_; }
  void increment (int inc) { n_ += inc; }

  auto make_lambda() const {
    return [*this]() mutable {
      this->increment_by(42);
      return this->number();
    };
  }
};
```

```cpp
auto make_lambda() const {
  return [copy_A=*this]() mutable {
    copy_A.increment_by(42);
    return copy_A.number();
  };
}
```

Being explicit with this

```
class A{
  int n_{0};

public:
  int number() const { return n_; }
  void increment (int inc) { n_ += inc; }

  auto make_lambda() const {
    return [*this]() mutable {
      this->increment_by(42);
      return this->number();
    };
  }
};
```

✗

Compile does not

```
auto make_lambda() const {
  return [copy_A=*this]() mutable {
    copy_A.increment_by(42);
    return copy_A.number();
  };
}
```

✓

Compiles

Being explicit with this

```
class A{
  int n_{0};

public:
  int number() const { return n_; }
  void increment (int inc) { n_ += inc; }

  auto make_lambda() const {
    return [this]() mutable {
      this->increment_by(42);
      return this->number();
    };
  }
};
```

✗ Compile does not

```
  auto make_lambda() const {
    return [&ref_A=*this]() mutable {
      ref_A.increment_by(42);
      return ref_A.number();
    };
  }
```

✗ And this also not

Being explicit with this

```
class A{
  int n_{0};

public:
  int number() const { return n_; }
  void increment (int inc) { n_ += inc; }

  auto make_lambda() {
    return [this]() mutable {
      this->increment_by(42);
      return this->number();
    };
  }
};
```

```
auto make_lambda() {
  return [&ref_A=*this]() mutable {
    ref_A.increment_by(42);
    return ref_A.number();
  };
}
```

Cherry-picking member variables

```
struct kv_proxy {

  database_context db_context{/*~~~*/};

  auto add_record(record const& rec) {
    auto key{ generate_key() };
    store_update(db_context, key, rec);
    return [key, dbc_copy=db_context](){ return store_retrieve(dbc_copy, key);};
  }

};
```

Cherry-picking member variables

```
struct kv_proxy {
```

```
error: use of deleted function

'database_context::database_context(database_context const&)'
```

Cherry-picking move-only objects

```
struct kv_proxy {

  database_context db_context{/*~~~*/};

  auto add_record(record const& rec) {
    auto key{ generate_key() };
    store_update(db_context, key, rec);
    return [key, dbc=std::move(db_context)]()
      {
        return store_retrieve(dbc, key);
      };
   }
};
```

Moving and lifetime

```
auto some_function(){
  record rec{/*~~~*/};
  kv_proxy proxy{/*~~~*/};
  return proxy.add_record(rec);
}
```

OK proxy (or its part) is moved into a lambda, and that's fine

```
void some_other_function(kv_proxy& proxy, const std::vector<record>& records){
  for (auto const& rec: records)
    proxy.add_record( rec );
}
```

NO proxy (or its part) is moved into a lambda, but it's still needed

Rule #9:

Use different capture modes to support different lifetime requirements.

Moving conditionally with ref-qualified functions

```
auto kv_proxy::add_record(record const& rec) && {
  auto key{ generate_key() };
  store_update(db_context, key, rec);
  return [key, dbc=std::move(db_context)]{ return store_retrieve(dbc, key); };
}
```

will be called for rvalue kv_proxy objects

```
auto kv_proxy::add_record(record const& rec) & {
  auto key{ generate_key() };
  store_update(db_context, key, rec);
  return [key, &dbc_ref=db_context]{ return store_retrieve(dbc_ref, key); };
}
```

will be called for lvalue kv_proxy objects

Moving and lifetime

```
auto some_function(){
  record rec{/*~~~*/};
  kv proxy proxy{/*~~~*/};
  return std::move(proxy).add_record(rec);
}
```

OK proxy (or its part) is moved into a lambda, and that's fine

```
void some_other_function(kv_proxy& proxy, const std::vector<record>& records){
  for (auto const& rec: records)
    proxy.add_record( rec );
}
```

OK proxy (or its part) is captured by reference

There’s more …

So many database, so much code…

```
struct kv_proxy {

  database_context db_context{ };

  auto add_record(record const& rec) & {
    auto key{ generate_key() };
    store_update(db_context, key, rec);
    return [key, &dbc_ref=db_context]()
      {
        return store_retrieve(dbc_ref, key);
      };
  }
};
```

So many database, so much code…

```
template <typename DBContext>
struct kv_proxy {

  DBContext db_context{ };

  auto add_record(record const& rec) & {
    auto key{ generate_key() };
    store_update(db_context, key, rec);
    return [key, &dbc_ref=db_context]()
      {
        return store_retrieve(dbc_ref, key);
      };
  }
};
```

So many database, so much code…

```
template <>
struct kv_proxy<configurable_db>{

  configurable_db db_context{ };

  auto add_record(record const& rec, store_policy const& pol) & {
    auto key{ generate_key() };
    store_update(db_context, key, rec, policy);
    return [key, &dbc_ref=db_context, pol]()
      {
        return store_retrieve(dbc_ref, key, pol);
      };
  }
};
```

So many database, so much code…

```
template <>
struct kv_proxy<secure_db> {

  secure_db db_context{ };

  auto add_record(record const& rec, store_policy const& pol, auth_token token) & {
    auto key{ generate_key() };
    store_update(db_context, key, rec, pol, token);
    return [key, &dbc_ref=db_context, pol, token=std::move(token)]()
      {
        return store_retrieve(dbc_ref, key, pol, token);
      };
  }
};
```

Rule #10:

Use parameter packs to store multiple objects in generic code.

Parameter...packs

```
auto consume(std::vector<int> a);
auto consume(std::string const& a, int b);
auto consume(int a, int b, int c);


auto take_many_arguments(      ? ? ?       ){

  return consume( ? ? ? );

}
```

Parameter...packs

```cpp
auto consume(std::vector<int> a);
auto consume(std::string const& a, int b);
auto consume(int a, int b, int c);

template <typename...Args>
auto take_many_arguments(Args const&...args){

  return consume(args...);

}

take_many_args(1, 2, 3);
take_many_args("answer", 42);
```

Parameter pack expansion

Capturing parameter...packs

```
template <typename DBContext>
struct kv_proxy {

  DBContext db_context{ };

  template <typename...StoreArgs>
  auto add_record(record const& rec, StoreArgs const&...sargs) & {
    auto key{ generate_key() };
    store_update(db_context, key, rec, sargs...);
    return [key, &dbc_ref=db_context, sargs...]()
      {
        return store_retrieve(dbc, key, sargs...);
      };
  }
};
```

Capture of a parameter pack (by copy)

Parameter pack expansion

Capturing parameter…packs

```
kv_proxy<json_store> proxy;

auto policy = storage_policy::default_json_policy();

/*~~~*/
auto auth = token::with_expiry(std::chrono::seconds{60});
proxy.add_record(some_record, policy, std::move(auth));
```

Capturing parameter...packs

```
template <typename DBContext>
struct kv_proxy {

  DBContext db_context{ };

  template <typename...StoreArgs>
  auto add_record(record const& rec, StoreArgs&&...sargs) & {
    auto key{ generate_key() };
    store_update(db_context, key, rec, sargs...);
    return [key, &dbc_ref=db_context, ...sargs=std::forward<StoreArgs>(sargs)]()
      {
        return store_retrieve(dbc, key, sargs...);
      };
  }
};
```

&& stands for forwarding reference

Capturing parameter...packs

```
kv_proxy<json_store> proxy;

auto policy = storage_policy::default_json_policy();

/*~~~*/
auto auth = token::with_expiry(std::chrono::seconds{60});
proxy.add_record(some_record, policy, std::move(auth));
```

policy will be copied into a lambda

auth will be moved into a lambda

Secret Rule #11:

Have fun with lambdas!

Time For Answers

Dawid Zalewski
github.com/zaldawid
zaldawid@gmail.com
saxion.edu