Chapter 6: Lambda Expressions

closure class
- A closure class is a class from which a closure is instantiated.
- Each lambda causes compilers to generate a unique closure class.
- The statements inside a lambda become executable instructions in the member functions of its closure class.

lambdaの文法
https://en.cppreference.com/w/cpp/language/lambda
[captures] <tparams>(C++20) (params) specifiers exception attr -> ret requires(C++20) { body }
[captures] (params) -> ret { body }
[captures] (params) { body }
[captures] { body }

capturesについて
- A comma-separated list of zero or more captures, optionally beginning with a capture-default.
- A lambda expression can use a variable without capturing it if the variable
  - is a non-local variable or has static or thread local storage duration (in which case the variable cannot be captured). thread local storage durationは分かっていないが。
  - is a reference that has been initialized with a constant expression.
- A lambda expression can read the value of a variable without capturing it if the variable
  - has const non-volatile integral or enumeration type and has been initialized with a constant expression
  - is constexpr and has no mutable members.

captureするのは何をしている?

captureはlambdaが使っている外部変数はまだ有効かを見ている?

captureは少なくとも外部変数の扱い方を決める。by-valueで使うか、by-referenceで使うか。

retについて
- Return type.
- If not present it's implied by the function return statements (or void if it doesn't return any value)

Item 31: Avoid default capture modes

by-reference captureの定義: [&]
- A by-reference capture causes a closure to contain a reference to a local variable or to a parameter that's available in the scope where the lambda is defined.
- If the lifetime of a closure created from that lambda exceeds the lifetime of the local variable or parameter, the reference in the closure will dangle.

by-value captureもdangleできる: [=]

例えばcopy pointers.

Captureができる条件:

Captures apply only to non-static local variables (including parameters) visible in the scope where the lambda is created.
- In the body of Widget::addFilter, divisor is not a local variable, it's a data member of the Widget class.

copy pointersの例: this

```
class Widget {
public:
        void addFilter() const;
private:
        int divisor;
```

```
};
void Widget::addFilter() const {
        filters.emplace_back(
                [=](int value) { return value % divisor == 0; }
        );
}
```

copyされるのはthisポインタです！ つまり、compilerの翻訳は以下：

```
void Widget::addFilter() const {
        auto currentObjectPtr = this;
        filters.emplace_back(
                [currentObjectPtr](int value)
                { return value % currentObjectPtr->divisor == 0; }
        );
}
```

だからもしcurrent Widget objectが壊されたら、このlambdaもdangleする！

- Understanding this is tantamount to understanding that the viability of the closures arising from this lambda is tied to the lifetime of the Widget whose this pointer they contain a copy of.
- default capture modeを使わなく、explicit captureを使いましょう。
  - A default capture mode is what made it possible to accidentally capture this when you thought you were capturing divisor in the first place.

C++14のもっと良いcapture: generalized lambda capture

```
void Widget::addFilter() const {
        filters.emplace_back(
                [divisor = divisor](int value)    // copy divisor to closure
                { return value % divisor == 0; }
        );
}
```

default by-value captures ([=])のもう1つディメリットは、使えるけどcaptureできない変数（例えば、objects with static storage duration）がcaptureできると見えること。

・Default by-reference capture can lead to dangling references.

・Default by-value capture is susceptible to dangling pointers (especially this), and it misleadingly suggests that lambdas are self-contained.

Item 32: Use init capture to move objects into closures

Object that's expensive to copy but cheap to move (e.g., most containers in the Standard Library).
Move-only object (e.g., a std::unique_ptr or a std::future)

C++14 supports moving objects into closures: init capture / generalized lambda capture

init captureの例： move a std::unique_ptr into a closure

```
auto pw = std::make_unique<Widget>();
auto func = [pw = std::move(pw)]
        { return pw->isValidated() && pw->isArchived(); };
```

- the scope on the left is that of the closure class.
- the scope on the right is the same as where the lambda is being defined.

```
auto func = [pw = std::make_unique<Widget>()]
        { return pw->isValidated() && pw->isArchived(); };
```

もOK。

std::bindを使ってinit captureを実現する

```
std::vector<double> data;     // object to be moved into closure
…
auto func =
        std::bind(
```

```
        []{const std::vector<double>& data)
        { /* uses of data */ },
        std::move(data)
    );
```

- Like lambda expressions, std::bind produces function objects.
  - The first argument to std::bind is a callable object.
  - Subsequent arguments represent values to be passed to that object.
- A bind object contains copies of all the arguments passed to std::bind.
  - For each lvalue argument, the corresponding object in the bind object is copy constructed.
  - For each rvalue, it's move constructed. （std::move(data)はちょうどrvalue）
- Emulating move-capture in C++11 consists of move-constructing an object into a bind object, then passing the move-constructed object to the lambda by reference.

Item 33: Use decltype on auto&& parameters to std::forward them

lambdaの4つoperator() : function call operator
https://en.cppreference.com/w/cpp/language/lambda
ClosureType::operator()(params)
1.ret operator()(params) const { body } (the keyword mutable was not used)
2.ret operator()(params) { body } (the keyword mutable was used)
3.template<template-params> (since C++14, generic lambda)
ret operator()(params) const { body }
4.template<template-params> //since C++14, generic lambda, the keyword mutable was used)
ret operator()(params) { body }

generic lambdas
- lambdas that use auto in their parameter specifications.
  - 実現（上記の4つoperator()を参考）： operator() in the lambda's closure class is a template

このItemのstory:

1.parameter specificationにautoを使える
auto f = [](auto x){ return normalize(x); };
2.rvalueを対応できるように、&&及びstd::forwardを使う
auto f = [](auto&& x){ return normalize(std::forward<???>(x)); };
???: There is a T in the templatized operator() inside the closure class generated by the lambda, but it's not possible to refer to it from the lambda.
対策：decltype(x), decltype almost always yields the type of a variable or expression without any modifications. (p30)
std::forward<T>及びstd::forward<T&&>は同じ（reference-collapsingのお陰で）。（Tはnon-reference type）
3.perfect-forwarding lambda
auto f = [](auto&& x){ return normalize(std::forward<decltype(x)>(x)); };

Item 34: Prefer lambdas to std::bind

C++14のstandard suffixes for seconds (s), milliseconds (ms), hours (h), etc.

```
auto setSoundL =
    [](Sound s)
    {
        using namespace std::chrono;
        using namespace std::literals;      // for C++14 suffixes
        setAlarm(steady_clock::now() + 1h,
                 s,
                 30s);
    };
```

std::bindがoverloadに対してfunction pointer typeのcastは必要：

```
using SetAlarm3ParamType = void(*)(Time t, Sound s, Duration d);
auto setSoundB =
        std::bind(static_cast<SetAlarm3ParamType>(setAlarm),
                std::bind(std::plus<>(),
                        std::bind(steady_clock::now),
                        1h),
                _1,
                30s);
```

・In C++14, there are no reasonable use cases for std::bind.

・In C++11 only, std::bind may be useful for implementing move capture or for binding objects with templatized function call operators.