Chapter 4: Smart Pointers

The reason why a raw pointers is hard to love:

1. whether it points a single object or to an arrayが分からない
    1. whether to use the single-object form (“delete”) or the array form (“delete []”)
2. whether the pointer owns the thing it points toが分からない
3. should you use delete, or is there a different destruction mechanism?
    1. difficult to ensure that you perform the destruction exactly once along every path in your code (including those due to exceptions).
4. whether the pointer dangles, i.e., points to memory that no longer holds the object the pointer is supposed to point to.

four smart pointers in C++11
std::auto_ptr (deprecated), std::unique_ptr, std::shared_ptr, std::weak_ptr

Item 18: Use std::unique_ptr for exclusive-ownership resource management

std::unique_ptrはraw pointerと効率的に同じ

- std::unique_ptrs are the same size as raw pointers.
    - If a raw pointer is small enough and fast enough for you, a std::unique_ptr almost certainly is, too.
    - 改善点：raw pointersを全部std::unique_ptrに変更する？CornerRoadとCornerLocalはfactoryにする？

std::unique_ptrの定義

- a non-null std::unique_ptr always owns what it points to.
- a move-only type
    - moving a std::unique_ptr transfers ownership from the source pointer to the destination pointer.
        - the source pointer is set to null.
    - copying a std::unique_ptr isn’t allowed.

std::unique_ptrのふさわしい応用：as a factory function return type

```
class Investment {…};
class Stock: public Investment {…};
class Bond: public Investment {…};
class RealEstate: public Investment {…};
template<typename… Ts>
std::unique_ptr<Investment> makeInvestment(Ts&&… params);
```

もしcustom deleterを使いたいなら：

```
auto delInvmt = [](Investment* pInvestment) // このlambdaはmakeInvestmentの中に移動する
                {
                        makeLogEntry(pInvestment);
                        delete pInvestment;
                };

template<typename… Ts>
std::unique_ptr<Investment, decltype(delInvmt)> makeInvestment(Ts&&… params);
// C++14だったら、autoでいい。少なくともcarlaのC++バージョンをチェックしよう！
// return type has size of Investment*
{
        std::unique_ptr<Investment, decltype(delInvmt)> pInv(nullptr, delInvmt);
        if ( /* a Stock object should be created */ )
         { pInv.reset(new Stock(std::forward<Ts>(params)…)); }
        else if ( /* a Bond object should be created */ )
```

```
        { pInv.reset(new Bond(std::forward<Ts>(params)…)); }
        else if ( /* a RealEstate object should be created */ )
        { pInv.reset(new RealEstate(std::forward<Ts>(params)…)); }
        return pInv;
}
```

- All custom deletion functions accepts a raw pointer to the object to be destroyed.
- Using a lambda expression to create delInvmt is more efficient than writing a conventional function.
- Attempting to assign a raw pointer (e.g., from new) to a std::unique_ptr won't compile.
    - reset is used to have pInv assume ownership of the object created via new.
- std::forwardは僕まだ分かっていない
- We'll be deleting a derived class object via a base class pointer.
    - For that to work, the base class - Investment - must have a virtual destructor.
- Deleters that are function pointers generally cause the size of a std::unique_ptr to grow from one word to two.
    - Stateless function objects (e.g., from lambda expressions with no captures) typically incur no size penalty.
    - When a custom deleter can be implemented as either a function or a captureless lambda expression, the lambda is preferable.
- std::unique_ptr<Investment, void (*)(Investment*)> makeInvestment(Ts&&… params); // return type has size of Investment* + at least size of function pointer!

std::unique_ptrの2つ形
std::unique_ptr<T> for individual objects
std::unique_ptr<T[]> for arrays // std::array, std::vector, and std::string are virtually always better data structure choices than raw arrays. raw arrayはやめましょう！
- there's no indexing operator (operator[]) for the single-object form.
- the array form lacks dereferencing operators (operator* and operator->).

std::unique_ptrがfactoryにふさわしいもう1つ理由：easily and efficiently converts to a std::shared_ptr.
- Factory functions can't know whether callers will want to use exclusive-ownership semantics for the object they return or whether shared ownership would be more appropriate.
- By returning a std::unique_ptr, factories provide callers with the most efficient smart pointer, but they don't hinder callers from replacing it with its more flexible sibling.

・Stateful deleters and function pointers as deleters increase the size of std::unique_ptr objects.

Item 19: Use std::shared_ptr for shared-ownership resource management

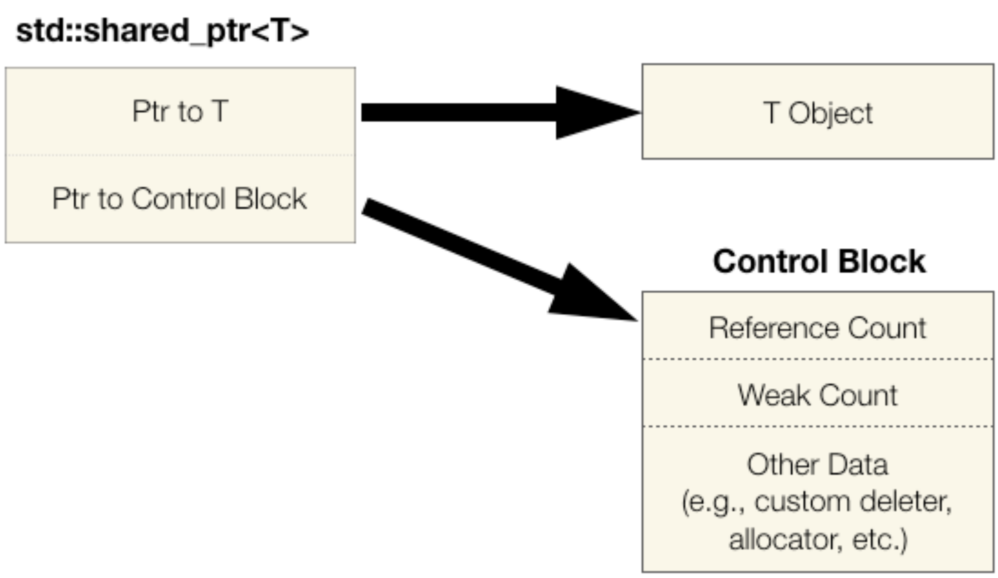
garbage collectionの制限：the only resource is memory and the timing of resource reclamation is nondeterministic.
resource管理の目標：a system that works automatically (like garbage collection), yet applies to all resources and has predictable timing (like destructors)
std::shared_ptrはこの目標に目指している。

std::shared_ptrの動作
- When the last std::shared_ptr pointing to an object stops pointing there (e.g., because the std::shared_ptr is destroyed or made to point to a different object), that std::shared_ptr destroys the object it points to.
- resource's reference count（なのでperformanceには影響する）
    - std::shared_ptrs are twice the size of a raw pointer
        - a raw pointer to the resource as well as a raw pointer to the resource's reference count
        - memory for the reference count must be dynamically allocated
            - pointed-to objects have no place to store a reference count.

    - the cost of the dynamic allocation is avoided when the std::shared_ptr is created by std::make_shared.
  - Increments and decrements of the reference count must be atomic.
    - reading and writing them is comparatively costly
    - Moving std::shared_ptr is faster than copying them: copying requires incrementing the reference count, but moving doesn't. (move-constructing & move-assignment)

std::shared_ptrとstd::unique_ptrの区別

deleterについて： For std::unique_ptr, the type of the deleter is part of the type of the smart pointer. For std::shared_ptr, it's not.
std::unique_ptr<Widget, decltype(loggingDel)> upw(new Widget, loggingDel)
std::shared_ptr<Widget> spw(new Widget, loggingDel);
そうすると、deleterが違うstd::shared_ptrでもタイプが同じになれる。



MEMORY ASSOCIATED WITH STD::SHARED_PTR<T> OBJECT

std::shared_ptr<T>と関連するメモリ
- An object's control block is set up by the function creating the first std::shared_ptr to the object.

control block creationに関するルール
- std::make_shared always creates a control block.
- A control block is created when a std::shared_ptr is constructed from a unique_ownership pointer (i.e., a std::unique_ptr or std::auto_ptr).
- When a std::shared_ptr constructor is called with a raw pointer, it creates a control block.
  - 悪い実装： Constructing more than one std::shared_ptr from a single raw pointer: the pointed-to object will have multiple control blocks.
    - Multiple control blocks means multiple reference counts, and multiple reference counts means the object will be destroyed multiple times.
  - 2 lesson
    - avoid passing raw pointers to a std::shared_ptr constructor.

        - use std::make_shared! しかしcustom deleterの場合はstd::make_sharedを使えない
        - raw pointers include this!
    - pass the result of the new directly.
        - std::shared_ptr<Widget> spw1(new Widget, loggingDel); // direct use of new
        - std::shared_ptr<Widget> spw2(spw1); // spw2 uses same control block as spw1

thisに対して、C++はstd::enable_shared_from_this base class templateを提供して、thisから std::shared_ptrを作れる

```
class Widget: public std::enable_shared_from_this<Widget> {
public:
	…
	void process();
	…
private:
	std::vector<std::shared_ptr<Widget>> processWidgets;
};
void Widget::process() {
	// as before, process the Widget
	…
	// add std::shared_ptr to current object to processWidgets
	processWidgets.emplace_back(shared_from_this()); // ようやくcarlaのgenerateWaypointsのコード
を理解した！ odr_mapに2つmapがあって、違和感が感じる。
}
```

carla/client/Map.cppのstd::vector<SharedPtr<Waypoint>> Map::GenerateWaypoints(double distance) constは正にこの実装だ！
https://github.com/carla-simulator/carla/blob/master/LibCarla/source/carla/client/Map.cpp

```
std::vector<SharedPtr<Waypoint>> Map::GenerateWaypoints(double
distance) const {
    std::vector<SharedPtr<Waypoint>> result;
    const auto waypoints = _map.GenerateWaypoints(distance);
    result.reserve(waypoints.size());
    for (const auto &waypoint : waypoints) {
    result.emplace_back(SharedPtr<Waypoint>(new
    Waypoint{shared_from_this(), waypoint}));
    }
    return result;
    }
```

https://github.com/carla-simulator/carla/blob/master/LibCarla/source/carla/client/Waypoint.h
carla/client/Waypoint.hにも確かにstd::enable_shared_from_thisから継承している！

```
class Waypoint
    : public EnableSharedFromThis<Waypoint>,
    private NonCopyable {
```

- std::enable_shared_from_this is a base class template.
    - Its type parameter is always the name of the class being derived, so Widget inherits from std::enable_shared_from_this<Widget>.

    - The Curiously Recurring Template Pattern (CRTP)
- shared_from_this looks up the control block for the current object, and it creates a new std::shared_ptr that refers to that control block
    - if the current object has no associated control block, behavior undefined!
    - だからclasses inheriting from std::enable_shared_from_this often declare their constructors private and have clients create objects by calling factory functions that return std::shared_ptrs.
    - carla::client::Waypointのconstructorは確かにprivateですが、factory functionは使っていない。

```
private:
    friend class Map;
    Waypoint(SharedPtr<const Map> parent,
    road::element::Waypoint waypoint);
```

std::shared_ptrの性能について
- control block makes use of inheritance, and there's even a virtual function.
- For the functionality they provide, std::shared_ptrs exact a very reasonable cost.
- Under typical conditions, where the default deleter and default allocator are used and where the std::shared_ptr is created by std::make_shared, the control block is only about three words in size, and its allocation is essentially free.
- Most of the time, using std::shared_ptr is vastly preferable to trying to manage the lifetime of an object with shared ownership by hand.
    - If you find yourself doubting whether you can afford use of std::shared_ptr, reconsider whether you really need shared ownership.
    - odr_mapは今carla::clientを使っていて、carla::clientは基本std::shared_ptrを使っている。本当にstd::shared_ptrを使う必要があるかチェックしよう！ つまり、本当にshared ownershipが必要かをチェックしよう！ 直接にcarla::roadのAPIを使ってもいい?

Item 20: Use std::weak_ptr for std::shared_ptr-like pointers that can dangle

std::weak_ptrの動作
- a pointer like std::shared_ptr that doesn't affect an object's reference count.
- track when it dangles, i.e., when the object it is supposed to point to no longer exists.
- std::weak_ptrs can't be dereferenced, nor can they be tested for nullness.
    - because std::weak_ptr isn't a standalone smart pointer.

auto spw = std::make_shared<Widget>();
std::weak_ptr<Widget> wpw(spw);
spw = nullptr; // RC goes to 0, and the Widget is destroyed. wpw now dangles.
if (wpw.expired()) … // if wpw doesn't point to an object

std::weak_ptrからstd::shared_ptrを作る
std::shared_ptr<Widget> spw1 = wpw.lock(); // if wpw's expired, spw1 is null
auto spw2 = wpw.lock(); // same as above
std::shared_ptr<Widget> spw3(wpw); // if wpw's expired, throw std::bad_weak_ptr

std::weak_ptrにふさわしい応用１： caching factory function
背景詳細：
1. Callers should certainly receive smart pointers to cached objects, and callers should certainly determine the lifetime of those objects // Callersはstd::shared_ptrを使う。
2. the cache needs a pointer to the objects, too, to detect when they dangle // Cacheはstd::weak_ptrを使う。

まだ改善しないといけないコードですが、caching factory function：

```
std::shared_ptr<const Widget> fastLoadWidget(WidgetID id) {
	static std::unordered_map<WidgetID,
					std::weak_ptr<const Widget>> cache;
	auto objPtr = cache[id].lock(); // objPtr is std::shared_ptr to cached object (or null if object's not in cache)
	if (!objPtr) {
		objPtr = loadWidget(id);
		cache[id] = objPtr;
	}
	return objPtr;
} // 改善できるのは、the cache may accumulate expired std::weak_ptrs corresponding to Widgets that are no longer in use (have therefore been destroyed).
```

std::weak_ptrにふさわしい応用２： the Observer design pattern

subjects (objects whose state may change) and observers (objects to be notified when state changes occur)

- each subject holds a container of std::weak_ptrs to its observers.

std::weak_ptrにふさわしい応用３： pointer cycle (A points to B & B points to A)

背景詳細:

- A and C share ownership of B and therefore hold std::shared_ptrs to it
- Suppose it'd be useful to also have a pointer from B back to A. What kind of pointer should this be?

3つ選択肢:

- raw pointer: if A is destroyed, B can't detect it → dereferencing A yield undefined behavior
- std::shared_ptr : デッドロックみたい。お互いにreference countを1にして、両方destroyされない。
- std::weak_ptr: A can be destroyed. B can detect it.

上記のstd::shared_ptr cycleはnot common!

- In strictly hierarchal data structures such as trees
  - Links from parents to children are thus generally best represented by std::unique_ptrs. 親が子ノードを独占するから。
  - Back-links from children to parents can be safely implemented as raw pointers, because a child node should never have a lifetime longer than its parent.

・Potential use cases for std::weak_ptr include caching, observer lists, and the prevention of std::shared_ptr cycles.

Item 21: Prefer std::make_unique and std::make_shared to direct use of new

理由1： no duplicate code

```
auto upw1(std::make_unique<Widget>()); // with make func
std::unique_ptr<Widget> upw2(new Widget); // without make func
```

a key tenet of software engineering: code duplication should be avoided.

理由2： no resource leak

without make funcのresource leakの場合

```
processWidget(std::shared_ptr<Widget>(new Widget), computePriority());
```

- At runtime, the arguments for a function must be evaluated before the function can be invoked.
- evaluationの1つ可能順番
  - Perform "new Widget"
  - Execute computePriority

  - Run std::shared_ptr constructor
- If such code is generated and, at runtime, computePriority produces an exception, the dynamically allocated Widget from Step 1 will be leaked, because it will never be stored in the std::shared_ptr that's supposed to start managing it in Step 3.

with make func、大丈夫!
processWidget(std::make_shared<Widget>(), computePriority());

理由3：with make func, 1回memory allocation。without，２回。

2回：one memory allocation for the Widget and a second allocation for the control block.

効果：reduce the static size of the program, increase the speed of the executable code.

make functionsの制限１：custom deleters対応できない

make functionsの制限2：braced initializer対応できない
- The make functions perfect-forward their parameters to an object's constructor
- braced initializers can't be perfect-forwarded

braced initializerの方を使いたいなら：
auto initList = {10, 20};
auto spv = std::make_shared<std::vector<int>>(initList);
make functionsの制限３（shared only）：
- Using make functions to create objects of types with class-specific versions of operator new and operator delete is typically a poor idea
  - 理由：The amount of memory that std::allocate_shared requests isn't the size of the dynamically allocated object, it's the size of that object plus the size of a control block.

make functionsの制限４（shared only）：
- The memory allocated by a std::shared_ptr make function can't be deallocated until the last std::shared_ptr and the last std::weak_ptr referring to it have been destroyed.
  - 理由：
    - std::shared_ptr's control block being placed in the same chunk of memory as the managed object.
    - std::weak_ptrさえ存在すれば、control blockのweak count0にならない。control blockはdestroyされない
  - 結果：If the object type is quite large and the time between destruction of the last std::shared_ptr and the last std::weak_ptr is significant, a lag can occur between when an object is destroyed and when the memory it occupied is freed.
  - newを使えば、この結果はない：With a direct use of new, the memory for the ReallyBigType object can be released as soon as the last std::shared_ptr to it is destroyed.つまりmemory for objectとmemory for control blockが分けられている。

newを使わないといけない場合は、exception-safety問題を対応しないといけない

解決策は、argumentsにnewを使わなく、newしたresultを渡す。

std::shared_ptr<Widget> spw(new Widget, cusDel);

processWidget(std::move(spw), computePriority()); // std::shared_ptrをcopyしないように、move
- Copying a std::shared_ptr requires an atomic increment of its reference count, while moving a std::shared_ptr requires no reference count manipulation at all.
- apply std::move to spw to turn it into an rvalue.

・Compared to direct use of new, make functions eliminate source code duplication, improve exception safety, and, for std::make_shared and std::allocate_shared, generate code that's smaller and faster.

・Situations where use of make functions is inappropriate include the need to specify custom deleters and a desire to pass braced initializers.

・For std::shared_ptrs, additional situations where make functions may be ill-advised include (1) classes with custom memory management and (2) systems with memory concerns, very large objects, and std::weak_ptrs that outlive the corresponding std::shared_ptrs.

Item 22: When using the Pimpl Idiom, define special member functions in the implementation file

なぜPimpl Idiom：

```
class Widget {
public:
	Widget();
	…
private:
	std::string name;
	std::vector<double> data;
	Gadget g1, g2, g3;
};
```

compile時間の課題：

- Widget clients must #include <string>, <vector>, and gadget.h, which increase the compilation time.
- these clients dependent on the contents of the headers.
  - If a header's content changes, Widget clients must recompile.

headersをincludeしないように、Pimpl Idiom：

Part 1: declaration:

```
class Widget {
public:
	Widget();
	~Widget();
	…
private:
	struct Impl;	// declare implementation struct and pointer to it.
	Impl *pImpl;
};
```

- Widget::Impl: incomplete type

Part 2: Implementation file

widget.cpp：recompileの課題一緒じゃない? compile時間変わるの?

```
// widget.h (visible to and used by Widget clients), widget.cpp (visible to and used by the Widget implementer)
#include "widget.h"
#include "gadget.h"
#include <string>
#include <vector>
struct Widget::Impl {
	std::string name;
	std::vector<double> data;
	Gadget g1, g2, g3;
};
Widget::Widget()
: pImpl(new Impl)
{}
Widget::~Widget()
{ delete pImpl; }
```

ステップ１：raw pointerをsmart pointerに入れ替える

ステップ２：implementation fileにdestructorを追加する

destructionに関するエラー

```
Widget w; // error!
```

原因：
- The issue arises due to the code that's generated when w is destroyed (e.g., goes out of scope).
- Prior to using delete, implementations typically have the default deleter employ C++11's static_assert to ensure that the raw pointer doesn't point to an incomplete type.

解決策：have the compiler see the body of Widget's destructor.

```
Widget::~Widget()
{}
```

defaultをつける（理由：the only reason you declared it was to cause its definition to be generated in Widget's implementation file.）

```
Widget::~Widget() = default;
```

ステップ３：moveを追加する。destructorのエラーと同じ解決策でimplementation fileにmoveのdefinitionも追加する！

ステップ４：copyを追加する。

custom copyを書かないといけない原因（default copy使えない原因）

1. compilers won't generate copy operations for classes with move-only types like std::unique_ptr.
2. even if they did, the generated functions would copy only the std::unique_ptr (shallow copy), we want to copy what the pointer points to (deep copy).

最終コード：

widget.h：

```
class Widget{
public:
        Widget();
        ~Widget();
        Widget(Widget&& rhs) noexcept;
        Widget& operator=(Widget&& rhs) noexcept;
        Widget(const Widget& rhs);
        Widget& operator=(const Widget& rhs);
        …
private:
        struct Impl;
        std::unique_ptr<Impl> pImpl;
};
```

widget.cpp：

```
#include "widget.h"
#include "gadget.h"
#include <string>
#include <vector>
struct Widget::Impl {
        std::string name;
        std::vector<double> data;
        Gadget g1, g2, g3;
};
Widget::Widget()
: pImpl(std::make_unique<Impl>())
{}
Widget::~Widget() = default;
Widget::Widget(Widget&& rhs) noexcept = default;
Widget& Widget::operator=(Widget&& rhs) noexcept = default;
Widget::Widget(const Widget& rhs)
: pImpl(nullptr)
```

```
{ if (rhs.pImpl) pImpl = std::make_unique<Impl>(*rhs.pImpl); }
Widget& Widget::operator=(const Widget& rhs) {
    if (!rhs.pImpl) pImpl.reset();
    else if (!pImpl) pImpl = std::make_unique<Impl>(*rhs.pImpl);
    else *pImpl = *rhs.pImpl;
    return *this;
}
```

ステップ５：（optional）もしstd::shared_ptrを使ったら？ std::unique_ptrはほぼ優先。

implementation fileにmoveやdeleterのdefinitionは全部いらなくなる。

理由：

- For std::shared_ptr, the type of the deleter is not part of the type of the smart pointer.
  - This necessitates larger runtime data structures and somewhat slower code, but pointed-to typed need not be complete when compiler-generated special functions are employed.
- std::unique_ptrの場合はほぼ逆。

・The Pimpl Idiom decreases build times by reducing compilation dependencies between class clients and class implementations.

・For std::unique_ptr pImpl pointers, declare special member functions in the class header, but implement them in the implementation file. Do this even if the default function implementations are acceptable.

・The above advice applies to std::unique_ptr, but not to std::shared_ptr.