Chapter 8: Tweaks

Item 41: Consider pass by value for copyable parameters that are cheap to move and always copied.

“alway copied”の意味は、std::move()が無条件で常に実施される。

move版関数とcopy版関数両方共用意すると、同じことをするけど、2つ関数になっちゃう。

inlineされたら大丈夫：Both functions will probably be inclined, and that’s likely to eliminate any bloat issues related to the existence of two functions, but if these functions aren’t inlined everywhere, you really will get two functions in your object code.

MapAPIにinlineで改善できるところある？ どんな関数をinlineにすべき？

template化にしても結局object codeに複数関数が生成されるかも

```
class Widget {
public:
	template<typename T>
	void addName(T&& newName) {
		names.push_back(std::forward<T>(newName));
	}
};
```

It may yield several functions in object code, because it not only instantiates differently for lvalues and rvalues, it also instantiates differently for std::string and types that are convertible to std::string.

解決方法：pass by value

```
void addName(std::string newName)
{ names.push_back(std::move(newName)); } // take lvalue or rvalue; move it
```

このやり方はCarlaのObjectのConstructorに結構使っている。pass by value and std::move it.

newNameをstd::moveする理由：

1. newName is a completely independent object from whatever the caller passed in, so changing newName won’t affect callers.
2. this is the final use of newName, so moving from it won’t have any impact on the rest of the function.

これは効率的にどう?

In C++11, however, newName will be copy constructed only for lvalues.
For rvalues, it will be move constructed.

```
std::string name(“Bart”);
w.addName(name); // call addName with lvalue
w.addName(name + “Jenne”); // call add Name with rvalue（operator+の実行結果だから）
```

pass by referenceの場合：The caller’s argument is bound to the reference newName. This is a no-cost operation. つまりpass by referenceの場合、call側が変数を関数に渡す時から関数が受け取るまで（関数実行前）、no-cost。だから今の実装の中に、pass by referenceのところ全然心配しなくて良い。心配すべきかもしれないところは関数内部どう扱うのだ。

効率分析

OverloadingとUsing a universal referenceの場合は、全部one copy for lvalues, one move for rvalues.

Passing by valueの場合は、one copy + one move for lvalues, two moves for rvalues.

move assignmentの場合はpass by value特に要注意

move assignmentの場合、もしpass by valueだと、an extra memory allocation（pass by valueのparameter） and deallocation（move assignmentの目的地の古いデータ）が必要になっちゃうかも。

- memory associated with the assignment target
- memory associated with the assignment source

void changeTo(std::string newPwd)
{ text = std::move(newPwd); } // pass by value, assign text
void changeTo(const std::string& newPwd)
{ text = newPwd; } // can reuse text's memory if text.capacity() >= newPwd.size()　しかし、text.capacity() < newPwd.size()の場合、どうしてもallocationとdeallocationが必要。pass by valueとpass by lvalueは差がない。

- This kind of analysis applies to any parameter type that holds values in dynamically allocated memory.
  - Not all types qualify, but many - including std::string and std::vector - do.
- Usually, the most practical approach is to adopt a "guilty until proven innocent" policy, whereby you use overloading or universal references instead of pass by value unless it's been demonstrated that pass by value yields acceptably efficient code for the parameter type you need.

pass by valueの危ないところ：the slicing problem

- If you have a function that is designed to accept a parameter of a base class type or any type derived from it, you don't want to declare a pass-by-value parameter of that type, because you'll "slice off" the derived-class characteristics of any derived type object that may be passed in.
- You'll find that the existence of the slicing problem is another reason (on top of the efficiency hit) why pass by value has a shady reputation in C++98.
- There are good reasons why one of the first things you probably learned about C++ programming was to avoid passing objects of user-defined types by value.

・For copyable, cheap-to-move parameters that are always copied, pass by value may be nearly as efficient as pass by reference, it's easier to implement, and it can generate less object code.

・For lvalue arguments, pass by value (i.e., copy construction) followed by move assignment may be significantly more expensive than pass by reference followed by copy assignment. rvalueは問題なさそう。copy constructionではなく、move constructionだから

・Pass by value is subject to the slicing problem, so it's typically inappropriate for base class parameter types.

Item 42: Consider emplacement instead of insertion.

std::forward_list
Compared to std::list this container provides more space efficient storage when bidirectional iteration is not needed. (https://en.cppreference.com/w/cpp/container/forward_list)
If you're not at all interested in performance, shouldn't you be in the Python room down the hall?

emplace_back does exactly what we desire: it uses whatever arguments are passed to it to construct a std::string directly inside the std::vector. No temporaries are involved.
vs.emplace_back("xyzzy"); // construct std::string inside vs directly from "xyzzy"

- emplace_back uses perfect forwarding
  - vs.emplace_back(50, 'x'); // insert std::string consisting of 50 'x' characters
- And every standard container that supports insert (which is all but std::forward_list and std::array) supports emplace. insertをemplaceに変更しよう！ 特にemplace(Function())!!点列を取る時まずcontainerに保存して、またmessageに詰め込む。2回コピーしてしまうだろう。
- The associative container offer emplace_hint to complement their insert functions that take a "hint" iterator, and std::forward_list has emplace_after to match its insert_after.

    - emplace_hintを持っているcontainerは：std::map, std::set, std::multimap, std::multiset, std::unordered_map, std::unordered_set, std::unordered_multimap, std::unordered_multiset, あとたくさんstd::pmr版のmap, set。つまりmapとsetはassociative containerだ。
- What makes it possible for emplacement functions to outperform insertion functions is their more flexible interface.
    - Insertion functions take objects to be inserted
    - Emplacement functions take constructor arguments for objects to be inserted
    - Emplacement can be used even when an insertion function would require no temporary, in that case, insertion and emplacement do essentially the same thing. つまりrvalue以外の場合?

insertion functionの方がrun fasterの場合
Such situations are not easy to characterize, because they depend on the types of arguments being passed, the containers being used, the locations in the containers where insertion or emplacement is requested, the exception safety of the contained types' constructors, and, for containers where duplicate values are prohibited (i.e., std::set, std::map, std::unordered_map, std::unordered_set), whether the value to be added is already in the container.

emplacement will almost certainly outperform insertionの条件：

1. The value being added is constructed into the container, not assigned.
    1. 例えば、vs.emplace(vs.begin(), "xyzzy"); // add "xyzzy" to beginning of vs. ほとんどのstd::vector実現にはmove assignmentが発生。constructは発生しない。
    2. Move assignment requires an object to move from, and that means that a temporary object will need to be created to be the source of the move.
    3. Node-based containers virtually always use construction to add new values, and most standard containers are node-based.
        1. The only ones that aren't are std::vector, std::deque, and std::string.
        2. Within the non-node-based containers, you can rely on emplace_back to use construction instead of assignment to get a new value into place, and for std::deque, the same is true of emplace_front.
2. The argument type(s) being passed differ from the type held by the container.
3. The container is unlikely to reject the new value as a duplicate.
    1. This means that the container either permits duplicates or that most of the values you add will be unique.
    2. The reason this matters is that in order to detect whether a value is already in the container, emplacement implementations typically create a node with the new value so that they can compare the value of this node with existing container nodes. move assignmentの場合、object to move fromが必要。detect duplicateの場合、object to compareが必要。違う。object to compareはemplaceが使っているConstructorが作る。object to move fromはそもそもconstructionではなく、assignmentだ。

emplacement functionを使う時もう一点注意：emplacement functionはexplicit constructorを使える。insertion functionはexplicit constructorを使えない。