Chapter 5: Rvalue References, Move Semantics, and Perfect Forwarding

perfect forwardingの定義：理解できるけど、なぜperfect forwardingしないといけないの？（下のstd::forwardの動作を参考できる）
make it possible to write function templates that take arbitrary arguments and forward them to other functions such that the target functions receive exactly the same arguments as were passed to the forwarding functions.

Item 23: Understand std::move and std::forward

std::move doesn't move anything.
std::forward doesn't forward anything.
At runtime, neither does anything at all.
They generate no executable code.

std::move and std::forward are merely functions (actually function templates) that perform casts.
std::move unconditionally casts its argument to an rvalue.
本当のmoveはmove constructorやmove assignment operatorがやっている？

std::moveの実現：argumentをrvalue referenceに変換する

```
template<typename T>
typename remove_reference<T>::type&&
move(T&& param)
{
        using ReturnType = typename remove_reference<T>::type&&;
        return static_cast<ReturnType>(param);
}
```

- remove_reference<T>の目的はensuring that "&&" is applied to a type that isn't reference.
  - That guarantees that std::move truly returns an rvalue reference.

C++14の実現

```
template<typename T>
decltype(auto)
move(T&& param)
{
        using ReturnType = remove_reference_t<T>&&;
        return static_cast<ReturnType>(param);
}
```

std::moveがcastしたrvalueがmoveされない場合：constが存在する時

```
class Annotation {
public:
        explicit Annotation(const std::string text)
        : value(std::move(text)) // textがconstなので、std::move(text)はrvalue const std::string
        { … }
…
private:
        std::string value;
};
```

stringのcopy ctorとmove ctorはこうなっている：

```
string(const string& rhs); // copy ctor
string(string&& rhs); // move ctor
```

- std::move(text)がconstなので、copy constructorを呼び出してしまう。
- 逆に、an lvalue-reference-to-const is permitted to bind to a const rvalue.

- don’t declare objects const if you want to be able to move from them.

もしかして、&&の目的はただoverload functionの選択用？つまり&&じゃないと、全部copy版のoverload関数を使っちゃう?

std::forwardの動作：std::forwardはconditional castをやっている。この部分大事!

例えば、lvalueやrvalue両方overloadした関数process：

```
void process(const Widget& lvalArg); // overload 1
void process(Widget&& rvalArg); // overload 2
template<typename T>
void logAndProcess(T&& param) {
        process(std::forward<T>(param));
}
Widget w;
```

logAndProcess(w); // wがlvalueなので、overload 1を呼び出したい!

logAndProcess(std::move(w)); // std::move(w)がrvalueなので、overload 2を呼び出したい!

しかし、paramは常にlvalueなので、std::forwardを使わないと、常にoverload 1を呼び出してしまう。

std::forwardの動作は、paramに渡したargument（wもしくはstd::move(w)）のタイプのままをparamのタイプにする。

- But param, like all function parameters, is an lvalue.
- Every call to process inside logAndProcess will thus want to invoke the lvalue overload for process.
- To prevent this, we need a mechanism for param to be cast to an rvalue if and only if the argument with which param was initialized - the argument passed to logAndProcess - was an rvalue.
- This is precisely what std::forward does.
- That’s why std::forward is a conditional cast: it casts to an rvalue only if its argument was initialized with an rvalue.

How std::forward can know whether its argument was initialized with an rvalue??

- brief answer: this information is encoded in logAndProcess’s template parameter T.

本当はstd::forwardだけでいいけど、std::moveがないと不便。

- std::move’s attractions are convenience, reduced likelihood of error, and greater clarity.

Item 24: Distinguish universal references from rvalue references

“T&&” has two different meanings!

1. rvalue references
2. universal references：何でもbindできる
    1. or forwarding references: universal references should almost always have std::forward applied to them.

universal referencesが出る2つ場合：共通点：type deductionが活躍している

1. function template parameters
    1. template<typename T>
       void f(T&& param);
2. auto declarations
    1. auto&& var2 = var1;

・If a function template parameter has type T&& for a deduced type T, or if an object is declared using auto&&, the parameter or object is a universal reference.

・If the form of the type declaration isn't precisely type&&, or if type deduction does not occur, type&& denotes an rvalue reference.
・Universal references correspond to rvalue references if they're initialized with rvalues. They correspond to lvalue references if they're initialized with lvalues.

Item 25: Use std::move on rvalue references, std::forward on universal references

質問：もしobjectのrvaluenessが分かったら、std::moveは必要? つまり、下記の例のrhsにstd::moveを適用する必要がる?
多分の答え：cast parameters bound to such objects to rvalues. つまりこのobjectにはstd::moveを適用しなくていいが、objectの例えばdata memberにstd::moveを適用する必要があるという意味? ? （多分違う）
多分の答え：関数parameterは常にlvalue（p3, p158）、だからparameterをstd::moveで変換する必要がある? ?

```
class Widget {
public:
        Widget(Widget&& rhs)
        : name(std::move(rhs.name)),
        p(std::move(rhs.p))
        { … }
private:
        std::string name;
        std::shared_ptr<SomeDataStructure> p;
};
```

Universal references should be cast to rvalues only it they were initialized with rvalues.
universal referenceにstd::forwardを使う悪い結果：can have the effect of unexpectedly modifying lvalues (e.g., local variables) つまりlocal variableをmoveするのが良くない。
例えば、

```
class Widget {
public:
        template<typename T>                    // universal reference compiles
        void setName(T&& newName)               // bad, bad, bad!
        { name = std::move(newName); }
private:
        std::string name;
        std::shared_ptr<SomeDataStructure> p;
};
std::string getWidgetName();
Widget w;
auto n = getWidgetName();               // n is local variable
w.setName(n);           // move n into w!
…                       // n's value now unknown
```

改善点：moveできそうなところを&&に付ける。例えば、"Crosswalk"など。

関数内何回&&のparameterを使う時、only final use of the referenceだけでstd::move/std::forwardを適用する。

If you're in a function that returns by value, and you're returning an object bound to an rvalue reference or a universal reference, you'll want to apply std::move or std::forward when you return the reference.

local variableをreturn valueにmoveするのはやめましょう!

理由１： このoptimizationは随分前から既に存在している。何もしなくてもこの最適化はやっている。
- It was recognized long ago that the “copying” version of makeWidget can avoid the need to copy the local variable w by constructing it in the memory allotted for the function’s return value.
- return value optimization (RVO)

理由２： the Standard requires that when the RVO is permitted, either copy elision takes place or std::move is implicitly applied to local objects being returned.
- The situation is similar for by-value function parameters.

・Apply std::move to rvalue references and std::forward to universal references the last time each is used.

・Do the same thing for rvalue references and universal references being returned from functions that return by value.

例えば：
```
Matrix
operator+(Matrix&& lhs, const Matrix& rhs) {
        lhs += rhs;
        return std::move(lhs);
}
```
・Never apply std::move or std::forward to local variables if they would otherwise be eligible for the return value optimization.

Item 26: Avoid overloading on universal references

改善点：local MapInfoの取得関数の中に絶対std::move/std::forwardを適用すべきところがある！copyしないように。例えばcontainerにemplace_backする時、copyしなくてもいいところがあるだろう？ emplace_backのparameterがlvalueの場合、copyは発生する。map messageを出来るだけ一回だけcopyして、その後全部移動しましょう！

一つuniversal referenceが効率を上げる例：
```
template<typename T>
void logAndAdd(T&& name) {
        …
        names.emplace(std::forward<T>(name));
}
std::string petName(“Darla”);
logAndAdd(petName);          // copy lvalue into multiset
logAndAdd(std::string(“Persephone”));        // move rvalue instead of copying it
logAndAdd(“Patty Dog”);      // create std::string in multiset instead of copying a temporary std::string
```
このuniversal reference関数に対して、overloadしない理由：
- Functions taking universal references are the greediest functions in C++. They instantiate to create exact matches for almost any type of argument.

perfect forwarding constructorの怖さ（greed）
```
class Person {
public:
        template<typename T>
        explicit Person(T&& n)
        : name(std::forward<T>(n)) {}

        Person(const Person& rhs); // copy ctor (compiler-generated)
        // p115: Member function templates never suppress generation of special member functions.
private:
        std::string name;
}
```

そうすると、copy constructorを呼び出さないことがある!
例えば:
Person p(“Nancy”);
auto cloneOfP(p);
なぜcloneOfP(p)がcopy constructorを呼び出さない?
- cloneOfP is being initialized with a non-const lvalue (p), and that means that the templatized constructor can be instantiated to take a non-const lvalue of type Person.
```
explicit Person(Person& n)
: name(std::forward<Person&>(n)) {}
```
- Calling the copy constructor would require adding const to p to match the copy constructor’s parameter’s type, but calling the instantiated template requires no such addition.

なので、const Person cp(“Nancy”); auto CloneOfP(cp);だったら、問題なくcopy constructorを呼び出せる!
- 理由: In situations where a template instantiation and a non-template function (a normal function) are equally good matches for a function call, the normal function is preferred.

・Overloading on universal references almost always leads to the universal reference overload being called more frequently than expected.
・Perfect-forwarding constructors are especially problematic, because they’re typically better matches than copy constructors for non-const lvalues, and they can hijack derived class calls to base class copy and move constructors.

Item 27: Familiarize yourself with alternatives to overloading on universal references

Use Tag dispatch

universal reference関数は1つ、関数内overloading。これは確かにCarlaに見たことがある。
https://github.com/carla-simulator/carla/blob/master/LibCarla/source/carla/road/Map.cpp
ForEachDrivableLane関数とForEachDrivableLaneImpl関数。tag dispatchではないかもしれないが
```
template<typename T>
void logAndAdd(T&& name) {
        logAndAddImpl(
                std::forward<T>(name),
                std::is_integral<std::remove_reference_t<T>>();       // C++14
        };
} // std::remove_reference_tが必要という理由は、もしlvalueのintが渡される場合、Tはint&になるので、std::is_integral<T>()（tags）はfalseを返す。
template<typename T>
void logAndAddImpl(T&& name, std::false_type) {    // non-integral argument
        auto now = std::chrono::system_clock::now();
        log(now, “logAndAdd”);
        names.emplace(std::forward<T>(name));
}
void logAndAddImpl(int idx, std::true_type) {          // std::true_type, std::false_type名前なし!
        logAndAdd(nameFromIdx(idx));         // std::string nameFromIdx(int idx);
        // avoid the need to put the logging code in both logAndAddImpl overloads.
}
```
Tag dispatch: It’s a standard building block of template metaprogramming.
- How it permits us to combine universal references and overloading.
- A keystone of tag dispatch is the existence of a single (unoverloaded) function as the client API.

Tag dispatchが処理できない状況: perfect-forwarding constructor

- Compilers may generate copy and move constructors themselves, so even if you write only one constructor and use tag dispatch within it, some constructor calls may be handled by compiler-generated functions that bypass the tag dispatch system.
- In truth, the real problem is not that the compiler-generated functions sometimes bypass the tag dispatch design, it's that they don't always pass it by.

Constraining templates that take universal references

std::enable_ifを使う!

- In our case, we'd like to enable the Person perfect-forwarding constructor only if the type being passed isn't Person.
- We need a way to strip any references, consts, and volatiles from T before checking to see if that type is the same as Person.
  - std::decay<T>::type (also turns array and function types into pointers)

文法：C++11

```
class Person {
public:
        template<typename T,
                typename = typename std::enable_if<condition>::type>
        explicit Person(T&& n);
        …
};
class Person {
public:
        template<typename T,
                typename = typename std::enable_if<
                !std::is_same<Person, typename std::decay<T>::type>::value>::type>
        explicit Person(T&& n);
        …
};
```

もう1つ課題：継承の場合

- The exact match (base class template) is better than the derived-to-base conversions that would be necessary to bind the SpecialPerson object to the Person parameters in Person's copy and move constructors.
- We want to enable the templatized constructor for any argument type other than Person or a type derived from Person.
- std::is_base_of<T1, T2>::valueを使う!
- User-defined types are considered to be derived from themselves, so std::is_base_of<T, T>::value is true if T is a user-defined type.
  - When T is a built-in type, std::is_base_of<T, T>::value is false.

```
class Person {
public:
        template<typename T,
                typename = typename std::enable_if<
                !std::is_base_of<Person, typename std::decay<T>::type>::value>::type>
        explicit Person(T&& n);
        …
};
```

C++14:（::typeいらない）

```
class Person {
public:
        template<typename T,
                typename = std::enable_if_t<
                !std::is_base_of<Person, std::decay_t<T>>::value>>
        explicit Person(T&& n);
        …
};
```

std::enable_ifはどうやってdistinguish integral and non-integral arguments?

```
class Person {
public:
        template<typename T,
                typename = std::enable_if_t<
                !std::is_base_of<Person, std::decay_t<T>>::value
                &&
                !std::is_integral<std::remove_reference_t<T>>::value
                >>
        explicit Person(T&& n) : name(std::forward<T>(n)) { … }
        explicit Person(int idx) : name(nameFromIdx(idx)) { … }
private:
        std::string name;
};
```

perfect forwardingのディメリット：error messageが出るのが遅い

static_assertを使う！解決していないが、分かれるメッセージは出れる。

```
explicit Person(T&& n) : name(std::forward<T>(n)) {
        // assert that a std::string can be created from a T object
        static_assert(
                std::is_constructible<std::string, T>::value,
                “Parameter n can’t be used to construct a std::string”
        );
… }
```

- Unfortunately, in this example the static_assert is in the body of the constructor, but the forwarding code, being part of the member initialization list, precedes it.

・Universal reference parameters often have efficiency advantages, but they typically have usability disadvantages.

Item 28: Understanding reference collapsing

universal reference template関数のencoding mechanism:

- When an lvalue is passed as an argument, T is deduced to be an lvalue reference.
  - つまりTは&、ではT&&は& &&になるでしょう。
  - なぜT&& paramが最終的に&（lvalue）になれる？reference collapsing!
- When an rvalue is passed, T is deduced to be a non-reference.（T&&だからでしょう）

&&はreference to referenceではない!

& &はreference to referenceです！illegal in C++!

4種類reference-reference:

- lvalue to lvalue
- lvalue to rvalue
- rvalue to lvalue
- rvalue to rvalue

reference collapsingのルール：

If either reference is an lvalue reference, the result is an lvalue reference.
Otherwise (i.e., if both are rvalue references) the result is an rvalue reference.

std::forwardが何をしている?

std::forwardの実現：

```
template<typename T>
T&& forward(typename remove_reference<T>::type& param) {
        return static_cast<T&&>(param);
}
```

C++14:

```
template<typename T>
T&& forward(typename remove_reference_t<T>& param) {
        return static_cast<T&&>(param);
}
```

lvalueが渡される場合、TはWidget&になる。

1. つまり、std::forwardは下記のようにinstantiateされる：

```
Widget& && forward(typename remove_reference<Widget&>::type& param) {
        return static_cast<Widget& &&>(param);
}
```

2. remove_reference<Widget&>::typeがWidgetになる：

```
Widget& && forward(Widget& param) {
        return static_cast<Widget& &&>(param);
}
```

3. reference collapsing

```
Widget& forward(Widget& param) {
        return static_cast<Widget&>(param);
}
```

rvalueが渡される場合、TはWidgetになる。

1. つまり、std::forwardは下記のようにinstantiateされる：

```
Widget&& forward(typename remove_reference<Widget>::type& param) {
        return static_cast<Widget&&>(param);
}
```

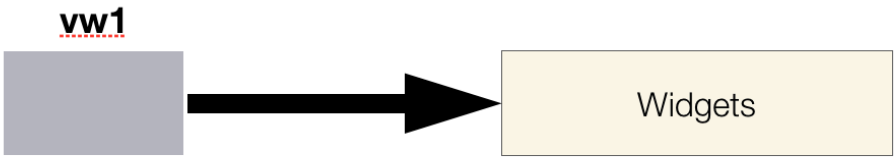
2. remove_reference<Widget>::typeがWidgetになる：

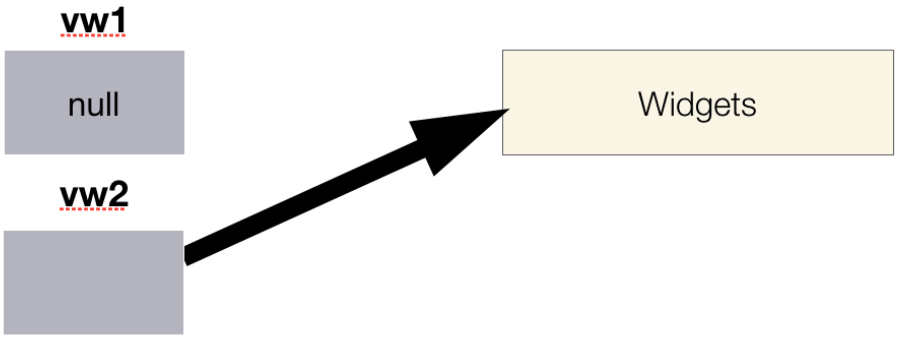
```
Widget&& forward(Widget& param) {
        return static_cast<Widget&&>(param);
}
```

・Reference collapsing occurs in four contexts: template instantiation, auto type generation, creation and use of typedefs and alias declaration, and decltype.

Item 29: Assume that move operations are not present, not cheap, and not used

std::vectorとstd::arrayのmove





STD::VECTOR AND ITS MOVE

STD::ARRAY AND ITS MOVE

- move std::vector: move vw1 into vw2. Runs in constant time. Only ptrs in vw1 and vw2 are modified.
- move std::array: move aw1 into aw2. Runs in linear time. All elements in aw1 are moved into aw2.

std::array以外、std::stringのmoveもcopyより早くないかも。
- Many string implementations employ the small string optimization (SSO).
- With the SSO, “small” strings (e.g., those with a capacity of no more than 15 characters) are stored in a buffer within the std::string object; no heap-allocated storage is used.
- The motivation for the SSO is extensive evidence that short strings are the norm for many applications.

Item 30: Familiarize yourself with perfect forwarding failure cases

direct callとindirect callの区別（perfect forwardingが失敗可能の原因）
- In a direct call to f, compilers see the arguments passed at the call site, and they see the types of the parameters declared by f.
  - They compare the arguments at the call site to the parameter declarations to see if they’re compatible, and, if necessary, they perform implicit conversions to make the call succeed.
- When calling f indirectly through the forwarding function template fwd, compilers no longer compare the arguments passed at fwd’s call site to the parameter declarations in f.
  - Instead, they deduce the types of the arguments being passed to fwd, and they compare the deduced types to f’s parameter declarations.

Braced initializersの失敗原因（fwd({ 1, 2, 3 })）
- Passing a braced initializer to a function template parameter that’s not declared to be a std::initializer_list is decreed to be a “non-deduced context”.
- 対策：auto il = { 1, 2, 3 }; fwd(il); // autoのため、il’s type deduced to be std::initializer_list<int>

Declaration-only integral static const and constexpr data members
意味：
class Widget {
public:

```
        static constexpr std::size_t MinVals = 28;
        …
};
…       // no definition for MinVals. MinValsが定義なしというのはWidget objectなし? いいえ、ただcppファイルに定義してない。
std::vector<int> widgetData;
widgetData.reserve(Widget::MinVals);
```

- Compilers work around the missing definition by plopping the value 28 into all places where MinVals is mentioned.
  - no storage
  - link失敗の場合：If MinVals' address were to be taken, then MinVals would require storage (so that the pointer had something to point to), and the code, though it would compile, would fail at link-time until a definition for MinVals was provided.
  - perfect forwardingはreferenceを扱うので、つまりaddressが必要。

Overloaded function names and template names

例えば、

```
void f(int (*pf)(int)); // or void f(int pf(int))
int processVal(int value);
int processVal(int value, int priority);
f(processVal); // fine
fwd(processVal); // error! which processVal?
```

- fwd, being a function template, doesn't have any information about what type it needs, and that makes it impossible for compilers to determine which overload should be passed.

function templateをforwardする時も同じ問題：

```
template<typename T>
T workOnVal(T param)
{ … }
fwd(workOnVal); // error! which workOnVal instantiation?
```

対策：

```
using ProcessFuncType = int (*)(int);
ProcessFuncType processValPtr = processVal; // specify needed signature for processVal
fwd(processValPtr);
fwd(static_cast<ProcessFuncType>(workOnVal));
```

Bitfields

Bitfieldsの例：

```
struct IPv4Header {
        std::uint32_t version:4,
                IHL:4,
                DSCP:6,
                ECN:2,
                totalLength:16;
        …
};
void f(std::size_t sz);
IPv4Header h;
fwd(h.totalLength); // error!
```

- The problem is that fwd's parameter is a reference, and h.totalLength is a non-const bitfield.
  - A non-const reference shall not be bound to a bit-field.
  - 理由：Bitfields may consist of arbitrary parts of machine words (e.g., bits 3-5 of a 32-bit int), but there's no way to directly address such things.
  - C++ dictates that the smallest thing you can point to is a char.

対策：

```
auto length = static_cast<std::uint16_t>(h.totalLength);
fwd(length);
```

・Perfect forwarding fails when template type deduction fails or when it deduces the wrong type.

・The kinds of arguments that lead to perfect forwarding failure are braced initializers, null pointers expressed as 0 or NULL, declaration-only integral const static data members, template and overloaded function names, and bitfields.