Introduction

C++11's most pervasive feature: move semantics.
rvalues (eligible for move operations)
- temporary objects returned from functions.

lvalues
- objects you can refer to, either by name or by following a pointer or lvalue reference.

dereference
Smart pointers normally overload the pointer-dereferencing operators (operator-> and operator*) ?

Chapter 2: auto

Item 6: Use the explicitly typed initializer idiom when auto deduces undesired types

- std::vector<bool>::reference is an example of a proxy class: a class that exists for the purpose of emulating and augmenting the behavior of some other type.
  - the Standard Library's smart pointer (std::shared_ptr, std::unique_ptr) types are proxy classes that graft resource management onto raw pointers.
  - smart pointerはinvisible proxy class typeではない。

- As a general rule, "invisible" proxy classes don't play well with auto.
  - Objects of such classes are often not designed to live longer than a single statement, so creating variables of those types tends to violate fundamental library design assumptions.
  - だから、下記のようなコードは避けたい。
    - auto someVar = expression of "invisible" proxy class type;

- Paying careful attention to the interfaces you're using can often reveal the existence of proxy classes. header filesを見よう。

- explicitly typed initializerの意味：
  - auto highPriority = static_cast<bool>(features(w)[5]);

Chapter 3: Moving to Modern C++

Item 7: Distinguish between () and {} when creating objects

Braced Initializationのメリット：

- Braced initialization (Uniform Initialization), 例えば
  - std::vector<int> v{ 1, 3, 5 };

- Uncopyable objects (e.g., std::atomic) may be initialized using braces or parentheses, but not using "="
  - std::atomic<int> ai1{ 0 };
  - std::atomic<int> ai2(0);

- A novel feature of braced initialization is that it prohibits implicit narrowing conversions among built-in types.
  - Initialization using parentheses and "=" doesn't check for narrowing conversions, because that could break too much legacy code. 面白い理由。

- Braced initialization's immunity to C++'s most vexing parse.
  - C++'s most vexing parseの意味：When we want to default-construct an object, but inadvertently end up declaring a function instead.

- Widget w2(); // Widgetのdefault constructorを呼び出したいが、w2という関数を宣言しちゃった。
- 解決方法：Widget w3{}; {}で関数を宣言できないから、Widgetのdefault constructorを呼び出す。

Braced Initializationの不便：
- When an auto-declared variable has a braced initializer, the type deduced is std::initializer_list.

- One or more constructors declare a parameter of type std::initializer_list, calls using the braced initialization syntax strongly prefer the overloads taking std::initializer_list.
  - 悪い例１：

```
class Widget {
public:
        Widget(int i, bool b);
        Widget(int i, double d);
        Widget(std::initializer_list<long double> il);
};
Widget w2{10, true};
Widget w4{10, 5.0};
// Widgets w2 and w4 will be constructed using the new constructor, even though the type of the std::initializer_list elements (long double) is, compared to the non-std::initializer_list constructors, a worse match for both arguments!
```

  - 悪い例２：

```
上記のstd::initializer_listのelement typeがboolになっても：
Widget(std::initializer_list<bool> il);
Widget w{10, 5.0}; // error! requires narrowing conversions (prohibited inside braced initializers)
```

  - normal overload resolutionが呼び出される場合：

```
Widget(std::initializer_list<std::string> il);
Widget w4{10, 5.0};
```

- Empty braces mean no arguments, not an empty std::initializer_list.

```
Widget w2{}; // also calls default constructor
Widget w4({}); // calls std::initializer_list constructor with empty list
```

- An example of where the choice between parentheses and braces can make a significant difference is creating a std::vector<numeric type> with two arguments.
  - std::vector<int> v1(10, 20);
    - use non-std::initializer_list constructor: create 10-element std::vector, all elements have value of 20
  - std::vector<int> v2{10, 20};
    - use std::initializer_list constructor: create 2-element std::vector, element values are 10 and 20

- Choosing between parentheses and braces for object creation inside templates can be challenging.

Item 8: Prefer nullptr to 0 and NULL

nullptrの定義：
- nullptr's advantage is that it doesn't have an integral type.
- It doesn't have a pointer type, either, but you can think of it as a pointer of all types.
- nullptr's actual type is std::nullptr_t, and, in a wonderfully circular definition, std::nullptr_t is defined to be the type of nullptr.
- The type std::nullptr_t implicitly converts to all raw pointer types, and that's what makes nullptr act as if it were a pointer of all types.

改善点：template関数まだ足りない。(2019/10/5)
例えば、CrosswalkとStopLineを取得する関数はtemplate化する。
また、関数templateの使う場合もあると思う。
また、SearchDirection enumもprevかnextを見ているところも適用できるだろう。

The fact that template type deduction deduces the “wrong” types for 0 and NULL (i.e., their true types, rather than their fallback meaning as a representation for a null pointer) is the most compelling reason to use nullptr instead of 0 or NULL when you want to refer to a null pointer.
- Avoid overloading on integral and pointer types

Item 9: Prefer alias declarations to typedefs

using STL containers is a good idea.
using std::unique_ptr is a good idea.
neither of us is fond of writing types like “std::unique_ptr<std::unordered_map<std::string, std::string>>” more than once. 改善点：僕正にこんなことをやってる！例えば、RoadMark style毎にlane pointsを取得する時使っている二重三重mapの長いタイプを何回も書いている！（2019/10/5）

alias declarationの例：
```
using UPtrMapSS = std::unique_ptr<std::unordered_map<std::string, std::string>>;
```

carla/LibCarla/source/carla/Memory.hにもalias宣言がある：
https://github.com/carla-simulator/carla/blob/master/LibCarla/source/carla/Memory.h
```
template <typename T>
using SharePtr = boost::shared_ptr<T>;
// templateの場合は正にtypedefではなく、alias宣言を使う理由。（alias templates）

template <typename T>
using WeakPtr = boost::weak_ptr<T>;

template <typename T>
using EnableSharedFromThis = boost::enable_shared_from_this<T>;
// boostを使う理由：Use this SharedPtr (boost::shared_ptr) to keep compatibility with boost::python, but it would be nice if in the future we can make a Python adaptor for std::shared_ptr.

template <typename T, typename… Args>
static inline auto MakeShared(Args &&… args) {
        return boost::make_shared<T>(std::forward<Args>(args)…);
} // このコードはまだ理解できていない。
```

Compilerがalias templateに会う時
```
template<typename T>
using MyAllocList = std::list<T, MyAlloc<T>>;
```
- When compilers process the Widget template and encounter the use of MyAllocList<T>, they know that MyAllocList<T> is the name of a type, because MyAllocList is an alias template: it must name a type.
- MyAllocList<T> is thus a non-dependent type, and a typename specifier is neither required nor permitted.
- If you haven’t done any TMP (template metaprogramming), that’s too bad, because if you want to be a truly effective C++ programmer, you need to be familiar with at least the basics of this facet of C++.

typedefで実現するとdependent typeになっちゃう：
```
template<typename T>
struct MyAllocList {
        typedef std::list<T, MyAlloc<T>> type;
```

```
};
template<typename T>
class Widget {
private:
        typename MyAllocList<T>::type list;
};
```

- MyAllocList<T>::type is dependent on a template type parameter (T). Thus it is a dependent type, and C++'s rule is that the names of dependent types must be preceded by typename.
- <type_traits> headerにあるtype transformation toolsにもtypedefでの実現が存在している（C++11）
  - std::remove_const<T>::type // C++11: const T -> T
  - std::remove_const_t<T> // C++14 equivalent
  - std::remove_reference<T>::type // C++11: T&/T&& -> T
  - std::remove_reference_t<T> // C++14: equivalent
  - std::add_lvalue_reference<T>::type // C++11: T -> T&
  - std::add_lvalue_reference_t<T> // C++14 equivalent
  - C++11のバージョンは全部dependent type。C++14の方を使いましょう。C++11を使いたい時でも簡単にalias templateを宣言しましょう。改善点：std::remove_const_t, std::remove_reference_t, std::add_lvalue_reference_tを使ったほうが良いところあるかを見てみる。

Alias templates avoid the "::type" suffix and, in templates, the "typename" prefix often required to refer to typedefs.
C++14 offers alias templates for all the C++11 type traits transformations.

Item 10: Prefer scoped enums to unscoped enums

unscoped enums: leak names
scoped enums: enum classes
scoped enumsの2つ強いメリット

1. the reduction in namespace pollution
2. their enumerators are much more strongly typed

unscoped enumsがusefulかもしれない唯一の場合：std::tuple

```
using UserInfo =
        std::tuple<std::string, // name
                std::string,    // email
                std::size_t>;   // reputation
UserInfo uInfo;
…
auto val = std::get<1>(uInfo); // get value of field 1. field 1の意味が分からないといけない。

enum UserInfoFields { uiName, uiEmail, uiReputation };
auto val = std::get<uiEmail>(uInfo); // ah, get value of email field.
```

しかし、この場合でもunscoped enumsを使わない方がいい。

scoped enumsの実現方法：(C++14)

```
template<typename E>
constexpr auto // std::getの<>の中に使われるので、template argument.
                // has to produce its result during compilation. must be a constexpr function.
        toUType(E enumerator) noexcept
{
        return static_cast<std::underlying_type_t<E>>(enumerator);
}
enum class UserInfoFields { uiName, uiEmail, uiReputation };
auto val = std::get<toUType(UserInfoFields::uiEmail)>(uInfo);
```

改善点：僕も<RoadId, SectionId, LaneId> tupleを使って、std::get<0>のように数字を使っている。よくないけど、本当に上記C++14の実現にする必要がある？数字でもいい気がする。

Enumerators of scoped enums convert to other types only with a cast.
Both scoped and unscoped enums support specification of the underlying type. The default underlying type for scoped enums is int. Unscoped enums have no default underlying type.
Scoped enums may always be forward-declared. （default underlying typeがあるから）
Unscoped enums may be forward-declared only if their declaration specifies an underlying type.

Item 11: Prefer deleted functions to private undefined ones

copy constructor and the copy assignment operator

Copyingがundesirableの例: istreams and ostreams

- Copying istreams and ostreams is undesirable, because it's not really clear what such operations should do.
  - If an istream were to be copied, would that entail copying all the values that had already been read as well as all the values that would be read in the future?
  - The easiest way to deal with such questions is to define them out of existence. Prohibiting the copying of streams does just that.
- C++98のやり方：private undefined
  - private:
    basic_ios(const basic_ios&); // not defined
    basic_ios& operator=(const basic_ios&); // not defined
- C++11のやり方：deleted functions
  - public: // 理由：C++ checks accessibility before deleted status
    basic_ios(const basic_ios&) = delete;
    basic_ios& operator=(const basic_ios&) = delete;
- deleted functionsのメリット（private undefined functionsと比べて）
  - Deleted functions may not be used in any way, so even code that's in member and friend functions will fail to compile if it tries to copy basic_ios objects.
  - Private undefined functionsだと、such improper usage wouldn't be diagnosed until link-time.

deleted functions used as deleted overloads

```
bool isLucky(int number);
bool isLucky(char) = delete; // reject chars
bool isLucky(bool) = delete; // reject bools
bool isLucky(double) = delete; // reject doubles and floats
```

こうする理由：C++'s C heritage means that pretty much any type that can be viewed as vaguely numerical will implicitly convert to int.

prevent use of template instantiations that should be disabled

```
template<typename T>
void processPointer(T* ptr);
```

- void*やchar*のinstantiationを避けたい（特別なpointers）
  - void*: there is no way to dereference them, to increment or decrement them, etc.
  - char*: they typically represent pointers to C-style strings, not pointers to individual characters

```
template<>
void processPointer<void>(void*) = delete;
template<>
void processPointer<char>(char*) = delete;
```

// const void*, const char*, volatile void*, volatile char*, const volatile void*, const volatile char*または他のcharタイプwchar_t, char16_t, char32_tも一個ずつ=delete。

template specializations must be written at namespace scope, not class scope

```
class Widget {
public:
```

```
        …
        template<typename T>
        void processPointer(T* ptr)
        { … }
        …
};
template<>
void Widget::processPointer<void>(void*) = delete;
```

だからprivate undefined functionsは使えない: It doesn't work outside classes, it doesn't always work inside classes, and when it does work, it may not work until link-time.

Any function may be deleted, including non-member functions and template instantiations.

Item 12: Declare overriding functions override

```
class Base {
public:
        virtual void doWork();
        …
};
class Derived: public Base {
public:
        virtual void doWork();
        …
};
std::unique_ptr<Base> upb = std::make_unique<Derived>();
// create base class pointer to derived class object. 改善点：今実装しているCorner, CornerRoad, CornerLocalは、このようにしたい。dynamic_castを使いたくない。でも違う。今Derived classにある新しい関数を呼び出したいので、derived class pointerに変換しないと、呼び出せない。
…
upb->doWork(); // call doWork through base class ptr; derived class function is invoked.
```

C++11からのoverridingが発生する条件（新しい）：
The functions' reference qualifiers must be identical.
Member function reference qualifiersの例：

```
class Widget {
public:
        …
        void doWork() &; // applies only when *this is an lvalue
        void doWork() &&; // applies only when *this is an rvalue
};
Widget makeWidget(); // factory function (returns rvalue)
Widget w;                    // normal object (an lvalue)
w.doWork();                          // calls Widget::doWork &
makeWidget().doWork();       // calls Widget::doWork &&
```

他のC++98からのoverridingが発生する条件、例えば：
The parameter types, the constness, the return types and exception specifications of the base and derived functions must be identical.

Declare overriding functions override

```
class Derived: public Base {
public:
        virtual void mf1() override;
        virtual void mf2(unsigned int x) override;
        virtual void mf3() && override;
        virtual void mf4() const override;
};
```

overrideを使わないと欲しくない場合がある：derived class virtuals that are supposed to override base class functions, but don't, need not elicit compiler diagnostics.

C++のcontextual keywords: override and final
- In the case of override, it has a reserved meaning only when it occurs at the end of a member function declaration.

Member function reference qualifiers
- draw distinction for the object on which a member function is invoked, i.e., *this.
- 類似：const at the end of a member function declaration, which indicates that the object on which the member function is invoked (i.e., *this) is const.

lvalueとrvalueを使い分ける例：
```
class Widget {
public:
        using DataType = std::vector<double>;
        …
        DataType& data() & // for lvalue Widgets, return lvalue
        { return values; }
        DataType&& data() && // for rvalue Widgets, return rvalue
        { return std::move(values); }
        …
private:
        DataType values;
}
```
分けないと、つまりDataType& data() { return values; }
```
Widget w;
auto vals1 = w.data(); // copy w.values into vals1. copy referenceだけだろう?
Widget makeWidget();
auto vals2 = makeWidget().data();
```
- The Widget is the temporary object returned from makeWidget (i.e., an rvalue), so copying the std::vector inside it is a waste of time.
- It'd be preferable to move it, but, because data is returning an lvalue reference, the rules of C++ require that compilers generate code for a copy.

Item 13: Prefer const_iterators to iterators

The container member functions cbegin and cend produce const_iterators, even for non-const containers, and STL member functions that use iterators to identify positions (e.g., insert and erase) actually use const_iterators.
```
std::vector<int> values;
…
auto it = std::find(values.cbegin(), values.cend(), 1983);
values.insert(it, 1998);
```
上記のコードにconst iteratorを使うべき理由：this code never modifies what an iterator points to
削除も挿入もiteratorが指している中身は変更しない。

non-member version of begin, end, rbegin, etc. VS. member function version
- Through an oversight during standardization, C++11 added the non-member functions begin and end, but it failed to add cbegin, cend, rbegin, rend, crbegin, and crend. C++14 rectifies that oversight.
- In maximally generic code, prefer non-member version of begin, end, rbegin, etc., over their member function counterparts.

Item 14: Declare functions noexcept if they won't emit exceptions

C++11のexception policy
- During work on C++11, a consensus emerged that the truly meaningful information about a function's exception-emitting behavior was whether it had any.
- Black or white, either a function might emit an exception or it guaranteed that it wouldn't.
- This maybe-or-never dichotomy forms the basis of C++11's exception specifications, which essentially replace C++98's. (C++98-style exception specifications remain valid, but they're deprecated.)

noexcept statusの大切さ
- Callers can query a function's noexcept status, and the results of such a query can affect the exception safety or efficiency of the calling code.
- As such, whether a function is noexcept is as important a piece of information as whether a member function is const.
- Failure to declare a function noexcept when you know that it won't emit an exception is simply poor interface specification.

noexceptがdesirable場合の例（大事）
1. std::vector::push_back
   - C++98のpush_backのやり方：
     - the transfer was accomplished by copying each element from the old memory to the new memory, then destroying the objects in the old memory. 絶対安全。strong exception safety guarantee.
   - C++11のpush_backのやり方：
     - 出来るだけreplace the copying of std::vector elements with moves. risk of violating push_back's exception safety guarantee.
     - "move if you can, but copy if you must"
     - How can a function know if a move operation won't produce an exception? The answer is obvious: it checks to see if the operation is declared noexcept.
2. swap
   - Interestingly, whether swaps in the Standard Library are noexcept is sometimes dependent on whether user-defined swaps are noexcept
   - conditionally noexcept
     - template <class T, size_t N>
     - void swap(T (&a)[N], T (&b)[N]) noexcept(noexcept(swap(*a, *b)));

Correctness is more important
- You should declare a function noexcept only if you are willing to commit to a noexcept implementation over the long term.
  - If you declare a function noexcept and later regret that decision, your options are bleak.
- Most functions are exception-neutral.
  - Such functions throw no exceptions themselves, but functions they call might emit one.
  - Exception-neutral functions are never noexcept, because they may emit such "just passing through" exceptions.
- natural noexceptが大事。無理やりnoexceptを実現するのがpoor software engineering

I. Because there are legitimate reasons for noexcept functions to rely on code lacking the noexcept guarantee, C++ permits such code, and compilers generally don't issue warnings about it.
II. noexcept is part of a function's interface, and that means that callers may depend on it.
III. noexcept is particularly valuable for the move operations, swap, memory deallocation functions, and destructors.

Item 15: Use constexpr whenever possible

It's good that constexpr functions need not produce results that are const or known during compilation!

constexpr objects

- Values known during compilation are privileged.
  - They may be placed in read-only memory, for example, and, especially for developers of embedded systems, this can be a feature of considerable importance.
  - can be used in contexts where C++ requires an integral constant expression.
    - array sizes, integral template arguments (including lengths of std::array objects), enumerator values, alignment specifiers,…
    - declare it constexpr, because then compilers will ensure that it has a compile-time value

constexpr functions

- produce compile-time constants when they are called with compile-time constants.
  - If they're called with values not known until runtime, they produce runtime values.
  - This means you don't need two functions to perform the same operation, one for compile-time constants and one for all other values.

constexpr functionの例

例えば、配列のサイズは3^n。コンパイル時既に計算済みのが必要。

```
constexpr
int pow(int base, unsigned exp) noexcept
{ … }
constexpr auto numConds = 5;
std::array<int, pow(3, numConds)> results;
```

constexprの超強い例 (改善点：出来るだけconstexprを使いましょう。)

```
class Point {
public:
        constexpr Point(double xVal = 0, double yVal = 0) noexcept
        : x(xVal), y(yVal)
        {}

        constexpr double xValue() const noexcept { return x; }
        constexpr double yValue() const noexcept { return y; }

        void setX(double newX) noexcept { x = newX; } // C++11にできない。C++14できる
        void setY(double newY) noexcept { y = newY; }
private:
        double x, y;
};
constexpr Point p1(9.4, 27.7);
constexpr Point p2(28.8, 5.3);

constexpr
Point midpoint(const Point& p1, const Point& p2) noexcept
{
        return { (p1.xValue() + p2.xValue()) / 2,
                 (p1.yValue() + p2.yValue()) / 2 };
}
constexpr auto mid = midpoint(p1, p2);
```

- This is very exciting.
- The object mid, though its initialization involves calls to constructors, getters, and a non-member function, can be created in read-only memory!

- The traditionally fairly strict line between work done during compilation and work done at runtime begins to blur, and some computations traditionally done at runtime can migrate to compile time.
- The more code taking part in the migration, the faster your software will run.

C++14ではsettersもconstexprにできる：

```
constexpr void setX(double newX) noexcept { x = newX; } // C++11にできない。C++14できる
constexpr void setY(double newY) noexcept { y = newY; }
```

I. It's important to note that constexpr is part of an object's or function's interface.

Item 17: Understand special member function generation

move constructorの動作（Widget(Widget&& rhs);）
- move-construct each non-static data member of the class from the corresponding member of its parameter rhs
- also move-construct its base class parts
- a memberwise move consists of move operations on data members and base classes that support move operations, but a copy operation for those that don't.

move special member functionとcopy special member functionの区別や関連
- The two copy operations are independent
- The two move operations are not independent.
  - If you declare either, that prevents compilers from generating the other.
    - 理由：If you declare, say, a move constructor for you class, you're indicating that there's something about how move construction should be implemented that's different from the default memberwise move that compilers would generate.
- move operations won't be generated for any class that explicitly declares a copy operation.
- Declaring a move operation (construction or assignment) in a class causes compilers to disable the copy operations.
- C++11 does not generate move operations for a class with a user-declared destructor.
  - 理由はRule of Threeです。
  - Rule of Three: if you declare any of a copy constructor, copy assignment operator, or destructor, you should declare all three.