Effective Modern C++

Intro

p.2 Heuristic to determine whether an expression is a lvalue is to ask if you can take its address. If you can, it typically is. If you can't, it's usually an rvalue.

p.2 The type of an expression is independent of whether the expression is an lvalue or rvalue.

p.3 Eg. you can have an lvalue of an rvalue reference type.

```
Widget(Widget&& rhs);
```

rhs is an lvalue, it's perfectly valid to take rhs's address within Widget's move constructor. ⇒ All parameters are lvalues.

p.4 Exception safe offers at least basic exception safety guarantee, that is, even if an exception is thrown, no data structures are corrupted and no resources are leaked. The strong exception safety guarantee assures callers that if an exception arises, the state of the program remains as it was prior to the call.

Chapter 1

Deducing Types

Item 1. p. 10 Template type deduction

```
template<typename T>
void f(ParamType param);        // declare f(.)

f(expr);                        // call f(.)
```

The type deduced for T is dependent not just on the type of expr, but also on the form of ParamType. 3 cases:

- ParamType is a pointer or reference type (but not universal ref).
- ParamType is a universal reference.
- ParamType is neither a pointer nor a reference.

p.11 Case 1. ParamType is a pointer or reference (but not universal ref.)

1. If expr's type is a reference, ignore the reference part.
2. Then pattern-match expr's type against ParamType to deduce T.

p.13 Case 2. ParamType is a universal reference

In a function template taking a type parameter T, a universal reference's declared type is T&&, (like an rvalue reference).

1. If expr is an lvalue, both T and ParamType are deduced to be lvalue references.
2. If expr is an rvalue, the "normal" Case 1 rules apply.

p.14 Case 3. ParamType is neither a pointer nor a reference

- we're dealing with pass-by-value. Param is a new object.

1. If expr's type is a reference, ignore the reference part.
2. Ignore const and volatile.

p.15 `const char * const ptr = "Fun with pointers";`

p.15 Array types are different from pointer types, even though in many contexts, an array decays to a pointer to its first element.

Array Arguments

p.16 There is no such thing as a function parameter that's an array.

```
void myFunc(int param[])        // legal syntax
```

But the array declaration is treated as a pointer declaration, so myFunc declaration is equivalent to `void myFunc(int * param);`

p.16 Because array parameter declarations are treated as if they were pointer parameters, the type of an array that's passed to a template function by value is deduced to be a pointer type.

p.16 Although functions can't declare parameters that are truly arrays, they can declare parameters that are <u>references</u> to arrays.

p.17 Function arguments - Function types can decay into function pointers. Type deduction for arrays applies to type deduction for functions and their decay into function pointers.

Item 2 p.18 <u>Auto type deduction</u>

Auto type deduction is template type deduction (with the sole exception of when brace initialisers are used).

p.19 Auto plays the role of T in the template, and the type specifiers for the variable act as ParamType.

```
auto x = 27;
```

Eg. `const auto& rx = x;`

Think of the above as a template:

```
template<typename T>
void func_for_rx(const T& param);   // conceptual template

func_for_rx(x);                     // conceptual call
```

p.21

```
int x1 = 27;
int x2(27);
int x3 = {27};
int x4{27};
```

Four syntaxes, but only one result: an int with value 27.

p.21
```
auto x1 = 27;
auto x2(27);
```
Same meaning as before: declare a int variable with value 27.

```
auto x3 = {27};
auto x4{27};
```
Declare a variable of type std::initializer_list<int> containing a single element with value 27.

p.21 Special type deduction rule for auto. When the initialiser for an auto-declared variable is enclosed in braces, {}, the deduced type is std::initializer_list<T>. This doesn't happen in template type deduction.

p.22 Eg.
```
auto x = {11, 23, 9};   // x's type is std::initializer_list<int>
```

```
template <typename T>
void f(T param);

f({11, 23, 9});
// error! can't deduce T
```

```
template <typename T>
void f(std::initializer_list<T> initList);

f({11, 23, 9});
// T deduced as int and
// initList is std::initializer_list<int>
```

p.23 auto in a function return type or a lambda parameter implies template type deduction, (not auto type deduction).

Item 3 p.23 <u>Decltype</u>

Given a name or expression, decltype tells you the name's or expression's type. (With rare exceptions unlikely to be encountered).

p.24 In C++11, primary use for decltype is declaring function templates where the function's return type depends on its parameter types.

p.26 To use decltype deduction for a function's return type, use decltype(auto) specifier as return type. (Auto specifies that the type is to be deduced, and decltype says decltype rules should be used for during the deduction).

p.26 The use of decltype(auto) is not limited to function return types. It can also be convenient for declaring variables when you want to apply decltype type deduction rules to initialising expression.

Eg.
```
Widget w;
const Widget& cw = w;

auto myWidget1 = cw;              // myWidget1 type is Widget
decltype(auto) myWidget2 = cw;    // myWidget2 type is const Widget&
```

p.28 Example of decltype exception - Applying decltype to a name yields the declared type for that name. Names are typically lvalue expressions. For lvalue expressions more complicated than names, however, decltype generally ensures that the type reported is an lvalue reference.

p.29 int x = 0; decltype(x) is int, but decltype((x)) is int& !

Item 4 <u>Viewing deduced types</u>

Declare a class template that we don't define:

```
template<typename T>
class TD;
```

A compiler will show the type it has deduced for a variable x within an error message when calling TD:

```
TD<decltype(x)> xType;    // elicit error containing x's type
```

Chapter 2

Item 5 <u>Prefer auto to explicit type declarations</u>

p.38 auto variables have their type deduced from their initialiser, so they must be initialised. So auto prevents a host of uninitialised variable problems.

p.38 Because auto uses type deduction it can represent types known only to compilers, (eg. a local variable whose type is that of a closure).

p.40 auto avoids verbose variable declarations and "type shortcuts," (eg. std::vector<int> v; unsigned sz = v.size(); whereas the official return type of v.size() is std::vector<int>::size_type).

p.41 auto prevents unintentional type mismatches, (eg. remember that the key of a map should be const, so map elements are std::pair<const key, value> and not std::pair<key, value>).

Item 6 Use explicitly typed initialiser idiom when auto deduces undesired type

p.43 Ordinarily for a std::vector<T> object, calling operator [] returns a T&. However, C++ forbids references to bits and std::vector<bool> is specified to represent its bools in packed form, one bit per bool. Therefore, operator [] for std::vector<bool> returns a proxy class std::vector<bool>::reference that acts like a bool&.

p.43/
```
std::vector<bool> v{1, 3, 5}; // suppose v also has elements added

bool mybool1 = v[0];          // works fine

auto mybool2 = v[0];          // incorrect, may lead to undefined behaviour!

auto mybool3 = static_cast<bool>(v[0]); // do this instead
```

Chapter 3

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

p.50 Initialisation is not assignment.

```
Widget w1;  Widget w2 = w1;   // Copy construction not assignment
```

p.50 C++ 11 introduced "uniform initialisation" through the use of braces {}. Can even be used to specify the initial contents of a container `std::vector<int> v{1, 2, 3}` which was formerly inexpressible.

p.51 Braced initialisation prohibits implicit narrowing conversions amongst built-in types: if the value of an expression within a brace initialiser isn't guaranteed to be expressible by the type of object being initialised, compiler will complain. Eg. `double x, y, z; int sum{x+y+z};`

p.51 Braced initialisation is immune to C++'s "most vexing parse" whereby anything that can be parsed as a declaration must be interpeted as one. Eg. `Widget w();` declares a function instead of default constructor.

p.52 Major drawback of braced initialisation is auto-declared variables with braced initialiser being deduced as type `std::initializer_list`.

Constructor overload resolution:

p.52 If one or more class constructors declare a `std::initializer_list` parameter, calls using braced initialisation syntax strongly prefer the overload taking `std::initialization_list` if there is any way for compilers to construe such a call.

p.55 Edge case: suppose a class supports default construction and also supports `std::initializer_list` construction. Using an empty set of braces to construct an object will call default constructor. Empty braces means no arguments, not an empty `std::initializer_list`. If you want to call a constructor with an empty `std::initializer_list`, do it by making the empty braces a constructor arg:
`Widget w1({}); Widget w2{{}};`

p.56 Consider the issues of confusing
`std::vector<numeric type> v1(10, 20);`
and `std::vector<numeric type> v2{10, 20};`

Item 8 Prefer nullptr to 0 and NULL

p.58 Neither 0 nor NULL has a pointer type. 0 is an int and NULL is an integral type.

p.59
```
void f(int);
void f(void*);

f(0);       // calls f(int)
f(NULL);    // might not compile, otherwise calls f(int)
f(nullptr); // calls f(void*)
```

Item 9 Prefer alias declarations to typedefs

p.63
```
typedef std::unique_ptr<std::map<std::string, std::string>> UPtrMapSS;

using UPtrMapSS = std::unique_ptr<std::map<std::string, std::string>>;
```

p.63 Types for function pointers:

```
typedef void (*FP)(int, const std::string&);

using FP = void (*)(int, const std::string&);
```

p.64 Alias templates:

```
template<typename T>
using MyAllocList = std::list<T, MyAlloc<T>>;
```

whereas using typedefs instead would require nesting a typedef inside a templatised struct. This leads to also needing to use ::type and typename if used within another template.

p.66 Type transformations =

```
std::remove_const_t<T>
std::remove_reference_t<T>
std::add_lvalue_reference_t<T>
```

Item 10 Prefer scoped enums to unscoped enums

p.67 C++98 enum : `enum Colour {black, white, red};`

The names of the enumerators belong to the scope containing the enum. The fact that these enumerator names "leak" gives rise to the official term for this kind of enum, "unscoped enum."

p.67 C++11 enum : `enum class Colour {black, white, red};`

p.67,68 C++11 enums don't leak names and are referred to as "scoped or class enums." This helps to reduce namespace pollution. They are also much more strongly typed.

Eg. `Colour c = Colour::white;` // c = white won't work

p.68 There are no implicit conversions for enumerators in a scoped enum to any type. Must instead perform a static_cast explicitly, eg. `static_cast<int>(c);`

p.69,70 Scoped enums have underlying type of int by default. Can also be overridden, eg. `enum class Colour : std::uint32_t;` This means they can be forward declared more easily compared to unscoped enums which have no default underlying type. Forward declaring helps to reduce the need to recompile if there are subsequent changes to an enum's enumerators.

Item 11 Prefer deleted functions to private undefined ones

p.75 Eg. basic_ios (const basic_ios &) = delete;

Private functions may still be called by class member functions and friends. The undefined function would only cause an error at linking time. However, deleted functions may not be used in any way. So member functions and friends will cause a compile time failure if they try to use a deleted function.

p.75 By convention, deleted functions are declared public, not private. Some compilers will otherwise complain about accessibility which doesn't really affect whether a deleted function can be used.

p.76, 78 Any function may be deleted while only member functions may be private. This makes it easy to delete class member function template specialisations, which must be declared at namespace scope outside the class, to prevent access with undesirable parameter types. (As such template specialisations are outside the class, they cannot be made private).

Item 12 Declare overriding functions override

p.79 "Overriding" is when a derived class provides their own implementation of a virtual base class function. Both the derived and base class functions will have the same signature. "Overloading" is when a given function name has multiple signatures, each signature having a different number or type of parameters.

p.81 Because declaring derived class overrides is easy to get wrong, make explicit that a derived class function is supposed to override a base class version. Eg. virtual void func() override;

p.83 "Final" is a contextual keyword introduced in C++11. Applying final to a virtual function prevents it from being overridden in derived classes. Final may also be applied to a class, in which case the class is prohibited from being used as a base class.

p.83,84 Class member functions can be referenced qualified. This makes it possible to limit use of a member function to lvalues only or to rvalues only.

```
void doWork() &;     // * this must be an lvalue
void doWork() &&;    // * this must be an rvalue
```

Item 13 Prefer const_iterators to iterators

p.86 The standard practice of using const wherever possible dictates that you should use const_iterators any time you need an iterator, yet have no need to modify what the iterator points to.

p.88 In maximally generic code, prefer non-member versions of begin, end, rbegin, etc. over their member function counterparts.

Eg. `auto it = std::find(std::cbegin(c), std::cend(c), val);`

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

p.90 whether a function is noexcept is an important a piece of information as whether a member function is const.

p.91 In a noexcept function, optimisers need not keep the runtime stack in an unwindable state if an exception would propagate out of the function, nor must they ensure that objects in a noexcept function are destroyed in the inverse order of construction should an exception leave the function.

p.92 swap is a key component of many STL algorithm implementations. Its widespread use renders the optimisations that noexcept affords especially worthwhile. Moreover, the fact that swapping higher-level data structures can generally be noexcept only if swapping their lower-level constituents is noexcept should motivate you to offer noexcept swap functions whenever you can.

p.94 By default, all memory deallocation functions and all destructors (both user-defined and compiler-generated) are implicitly noexcept.

Item 15 Use constexpr whenever possible

p.97 constexpr objects are const and they have values that are known at compile time. Integral values that are constexpr are "integral constant expressions" and can be used to specify array sizes, integral template arguments, enumerator values and more.

p.98 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.

Item 16 Make const member functions thread safe

p.104 The mutable storage class is used only for class data-members, and allows those data-members to be modified even if they are members of const objects.

p.104 As part of caching activity, a const member function may need to modify cached values which are declared as mutable. However, this may mean that there is a data race if multiple threads attempt to read and write the same memory without synchronisation. (A const member function appears as a read only operation to the caller).

p.105 Employ a mutex when multiple variables or memory locations need to be synchronised.

```
mutable std::mutex m;
std::lock_guard<std::mutex> g(m);
```

p.105 If only a single variable or memory location needs synchronisation, an atomic variable may offer better performance than a mutex.

```
mutable std::atomic<unsigned> callcount{0};
```

p.105,106 Both std::mutex and std::atomic are uncopyable and unmovable, so the existence of such data members in a class means that the class is also neither copyable nor movable.

Item 17 Special member function generation

p.109 C++98 has four such functions: default constructor, the destructor, the copy constructor and the copy assignment operator. These functions are only generated if needed. A default constructor is generated only if the class declares no constructors at all. Generated special member functions are implicitly public and inline. They're non-virtual unless it's a destructor in a derived class inheriting from a base class with a virtual destructor.

p.109 C++11 introduced two more special member functions: move constructor and the move assignment operator.

```
Widget(Widget&& rhs);            // move constructor

Widget& operator=(Widget&& rhs); // move assignment operator
```

p.110 Unlike the two copy member functions which are independent, the two move operations are dependent - if you declare either, the other will not be automatically generated.

p.110 Furthermore, move operations won't be generated for any class that explicitly declares a copy operation. And vice-versa, declaring a move operation in a class causes compilers to disable the copy operations.

p.112 Move operations are generated for classes (when needed) only if these three things are true:

- No copy operations are declared in the class.
- No move operations are declared in the class.
- No destructor is declared in the class.

p.112 Note that base classes should have a virtual destructor declared. Often, the default implementation would be correct, and "= default" is a good way to express that. However, a user-declared destructor will suppress generation of the move operations. If these are then manually implemented, the copy operations will be suppressed.

p.112 To support all operations, quickly add:

```
virtual ~Base() = default;                          // Virtual destructor
Base(Base&&) = default;                             // Move construct
Base& operator=(Base&&) = default;                  // Move assign
Base(const Base&) = default;                        // Copy construct
Base& operator=(const Base&) = default;             // Copy assign
```

p.114 Full rules of C++11 governing special member functions.

Chapter 4 Smart Pointers

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

p.119 std::unique_ptr embodies exclusive ownership semantics. A non-null std::unique_ptr always owns what it points to.

p. 119 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.

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

p. 125 An object accessed via std::shared_ptrs has its lifetime managed by those pointers through shared ownership. No specific std::shared_ptr owns the object. When the last std::shared_ptr pointing to an object stops pointing there, that std::shared_ptr destroys the object it points to.

p. 133 If exclusive ownership will do or even may do, std::unique_ptr is a better choice. Its performance profile is close to that for raw pointers, and "upgrading" from std::unique_ptr to std::shared_ptr is easy, because a std::shared_ptr can be created from a std::unique_ptr. The reverse is not true.

p. 134 Compared to std::unique_ptr, std::shared_ptr objects are typically twice as big, incur overhead for control blocks, and require atomic reference count manipulations.

p. 134 Avoid creating std::shared_ptrs from variables of raw pointer type.

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

p. 134 std::weak_ptrs are typically created from std::shared_ptrs. They point to the same place as the std::shared_ptrs initialising them, but they don't affect the reference count of the object they point to.

p. 135 To check to see if a std::weak_ptr has expired and, if not, to gain access to the object it points to, create a std::shared_ptr from the std::weak_ptr.

p. 139 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.

p. 139 std::make_unique and std::make_shared are two of the three make functions. They take arbitrary sets of arguments, perfect-forward them to the constructor for a dynamically allocated object, and return a smart pointer to that object. std::allocate_shared acts just like std::make_shared, except its first argument is an allocator object used for the dynamic memory allocation.

p. 140 Using make functions reduces code duplication, avoids potential resource leaks and is more efficient when allocating memory for std::make_share and std::allocate_shared. (Only a single memory allocation is made for both the underlying object and associated control block rather than each being allocated seperately).

p. 143 Within the make functions, the perfect forwarding code uses parentheses, not braces. If you are trying to pass an initialiser list, there is a workaround:

```
auto initList = {10, 20};
auto spv      = std::make_shared<std::vector<int>>(initList);
```

p. 146 If you must use new (rather than a make function), make sure that you immediately pass the result to a smart pointer constructor in a statement that does nothing else. This prevents compilers from potentially generating code that emits an exception between the use of new and invocation of the constructor for the smart pointer that will manage the new created object.

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

p. 147 To combat excessive build times, use the "pointer to implementation" idiom. You replace the data members of a class with a std::unique_ptr to an implementation class (or struct), put the data members that used to be in the primary class into the implementation class, and access those data members indirectly through the pointer.

p. 156 Build times decrease by reducing compilation dependencies between class consumers and class implementations. (I.e. #include directives can be moved over to the source file).

p. 156 If using std::unique_ptr, 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. (This is unnecessary if using std::shared_ptr).

Chapter 5 Rvalue References, Move Semantics and Perfect Forwarding

p. 158 A parameter is always an lvalue, even if its type is an rvalue reference. That is, given void f(Widget&& w); the parameter w is an lvalue.

Item 23 Understanding std::move and std::forward

p. 158 std::move and std::forward are function templates that perform casts. std::move unconditionally casts its arguments to an rvalue, while std::forward performs this cast conditionally.

p. 161 Don't declare objects const if you want to be able to move from them. Move requests on const objects are silently transformed into copy operations. std::move doesn't guarantee that the object it's casting will be eligible to be moved.

p.161 The most common use of std::forward is in a function template taking a universal reference parameter that is to be passed to another function. std::forward casts its argument to an rvalue only if its argument was initialised with an rvalue.

Item 24 Distinguish universal references from rvalue references

p.164 To declare an rvalue reference to some type T, you write T&&. It thus seems reasonable to assume that if you see "T&&" in source code, you're looking at an rvalue reference - alas, it's not quite that simple.

p.164 T&& may be a universal / forwarding reference. Such references look like rvalue references in the source code but could be either a rvalue reference (permitting binding to rvalues) or a lvalue reference (permitting binding to lvalues). Furthermore, they can bind to const or non-const objects, or volatile or non-volatile objects.

p.164 Universal references arise in function template parameters and in auto declarations, ie. in contexts where type deduction occurs. If you see "T&&" without type deduction, you're looking at an rvalue reference.

p.165 For a reference to be universal, type deduction is necessary, but it's not sufficient. The form of the reference must be precisely "T&&". Hence in

```
template <typename T>
void f(std::vector<T>&& param);
```

param is an rvalue reference.

p.166 Even the presence of a const qualifier is enough to disqualify a reference from being universal:

```
template <typename T> void f(const T&& param);   // rvalue reference
```

p.167 Variables declared with the type auto&& are universal references, because type deduction takes place and they have the correct form ("T&&").

p.168 If the form of the type declaration isn't precisely T&&, or if type deduction does not occur, T&& denotes an rvalue reference.

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

p.170 Rather than writing two functions, one taking an lvalue reference and one taking a rvalue reference, it's better to write one function taking a universal reference.

p.172 If you're in a function that returns by value, and you're returning an object bound to an rvalue reference or universal reference, apply std::move or std::forward when you return the reference. This may allow the object to be moved into the return value location.

p.174 But don't try to optimise when a function returns by value and you're returning a local variable by applying std::move to the local variable in the return statement. It will already be optimised through "return value optimisation" (RVO). The local variable will be constructed in the memory alloted for the function's return value. std::move creates a reference to the local variable and the RVO can't be applied, so you will perform an unnecessary move.

p.177 Apply std::move to rvalue references and std::forward to universal references the last time each is used, (eg. perhaps when they are forwarded as arguments to functions, if the function call is the last act of the current function).

Item 26 Avoid overloading functions taking universal references

p. 183 Overloading on universal references almost always leads to the universal reference overload being called more frequently than expected because the universal reference can give an exact match on the argument provided by the caller.

p. 183 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 Alternatives to functions overloading on universal references

p. 184 Abandon overloading by using different function names.

p. 184 Pass by const T&. Won't be as efficient performance wise but avoids pitfalls.

p. 184 Pass by value when you know you'll end up copying the parameters anyway. (See Item 41).

p. 185 Tag dispatch design. The dispatching function is not overloaded. It takes an unconstrained universal reference parameter. The implementation functions are overloaded, and one takes a universal reference parameter, but resolution of calls to these functions depends not just on the universal reference parameter, but also on the tag parameter, and the tag values are designed so that no more than one overload will be a viable match.

p. 188 Constraining templates that take universal references. This can be done through std::enable_if.

C++11 code:

p. 189, 191

```
template <typename T,
          typename = typename std::enable_if<"condition">::type>
void func(T&& n){---};
```

and where "condition" is:

```
!std::is_same<"avoid type", typename std::decay<T>::type>::value
```

p. 192 But if we are dealing with a constructor function for a class, we should also consider the case where "avoid type" may also be a derived class, "condition" should be upgraded to:

```
!std::is_base_of<"avoid type", typename std::decay<T>::type>::value
```

p. 193 C++14 code:

```
template <typename T,
          typename = std::enable_if_t<"condition">>
void func(T&& n){---};
```

"condition": `!std::is_base_of<"avoid type", std::decay_t<T>>::value`

p. 195 When perfect-forwarding parameters to constructors, if the parameters mismatch what's required by the constructor, strange error messages are likely to result. The more complex the system, the more likely that a universal reference is forwarded through several layers of function calls before finally arriving at a site that determines whether the argument type(s) are acceptable.

p. 196 Use a static_assert to check a parameter's type is acceptable:

```
static_assert(std::is_constructable<"type being constructed", T>::value,
              "Parameter can't be used to construct type X");
```

Item28 Reference Collapsing

p. 197 When an argument is passed to a template function, the type deduced for the template parameter encodes whether the argument is an lvalue or an rvalue. But this only happens when the argument is used to initialise a universal reference parameter. When an lvalue is passed, T is an lvalue reference. When an rvalue is passed, T is a non-reference.

p. 199 Only compilers are allowed to produce references to references. Reference collapsing dictates what happens next. If either reference is an lvalue reference, the result is an lvalue reference. Otherwise, if both are rvalue references, the result is an rvalue reference.

p. 198 Eg. template<typename T> void func(T&& param);

func(lvalue) : T is lvalue reference, param is (lvalue &) &&.
func(rvalue) : T is non-reference, param is rvalue &&.

p. 203 Reference collapsing occurs in four contexts: template instantiation, auto type generation, creation and use of typedefs and alias declarations, and decltype.

Item29 Assume that move operations are not present, not cheap, and not used

p. 206 There are several scenarios in which C++11's move semantics do you no good.

- No move operations for object. Move request becomes a copy.
- Move operations are no faster than copy operations.
- Move not usable. The context of the move request requires a move operation that emits no exceptions, but noexcept isn't declared.
- Source object is an lvalue. With very few exceptions only rvalues may be used as the source of a move operation.

Item 30 Familiarise yourself with Perfect Forwarding failure cases.

p.207

```
template <typename T>
void fwd(T&& param){
...
f(std::forward<T>(param));
}
```

```
template <typename... Ts>
void fwd(Ts&&... params){
...
f(std::forward<Ts>(params)...);
}
```

p.208 Braced initialisers. Passing a braced initialiser to a function template parameter that's not declared to be a std::initializer_list is deemed to be a "non-deduced context". However auto can perform type deduction for variables initialised with a brace initialiser:

```
fwd({1,2,3});          // error! doesn't compile
```

p.209

```
auto il = {1,2,3};     // il's type deduced to be std::initializer_list<int>
fwd(il);               // fine. perfect forwards it to f
```

p.209 0 or NULL as null pointers. 0 and NULL will be deduced as int and integral type, respectively, instead of a pointer type. Use nullptr instead. (Item 8)

p.210 Declaration-only integral static const and constexpr data members. Universal references and references are essentially the same thing as a pointer, they must point to some memory. However, memory will only be allocated once a corresponding definition has been provided.

p.211 Overloaded function names and template names. To workaround this, you can create a function pointer of the required type, thus indicating the overloaded function version to select or the proper template instantiation to generate.

p.212 `using FuncType = int(*)(int);`

```
FuncType ptr = process;        // correct process version selected
                 overloaded function
fwd(ptr);

fwd(static_cast<FuncType>(work));
                          template function
```

p.213 Bitfields. Bitfields may consist of arbitrary parts of machine words (eg. bit 3-5 of a 32-bit int), but theres no way to directly address such things. References and pointers are essentially the same thing at the hardware level, and just as there's no way to create a pointer to arbitrary bits, there's no way to bind a reference to arbitrary bits, either.

p.214 The key to passing a bitfield into a perfectly-forwarding function, is to take advantage of the fact that the forwarded-to function will always receive a copy of the bitfields value.

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

Chapter 6 Lambda Expressions

p.215 Everything a lambda can do is something you can do by hand with a bit more typing.

p.215 A closure is the runtime object created by a lambda. Depending on the capture mode, closures hold copies of or references to the captured data.

p.216 `auto c1 = [x](int y){return x*y > 55;};`

[ ] : what's captured when the closure gets created.
( ) : what's passed when the closure gets called.

Item 31 Avoid default capture modes

p.216 There are two default capture modes, [&] by-reference and [=] by-value.

p.217 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. Even if a default by-reference capture, [&], were replaced by an explicit capture, [&localvar], the reference would still dangle, but it's easier to see that the viability of the lambda is dependent on localvar's lifetime.

p.219 It's simply better software engineering to explicitly list the local variables and parameters that a lambda depends on.

p.219 Default by-value capture isn't immune to dangling. Eg. if you capture a pointer by value, you copy the pointer into the closures arising from the lambda, but you don't prevent code outside the lambda from deleting the pointer and causing your copies to dangle.

p.220 Captures apply only to non-static local variables, including parameters, visible in the scope where the lambda is created. (I.e. even when capturing by-value).

p.220 Every non-static class member function has a "this" pointer, and you use that pointer every time you mention a data member of the class.

p.220, 221 Thus, when a lambda using a class data member, is defined within a member function, even with a default by-value capture, what's being captured is not the required data member, but rather the class' "this" pointer. Understanding this is tantamount to understanding that the viability of the closures arising from this lambda is tied to the lifetime of the class' "this" pointer.

p.222 This particular problem can be solved easily by making a local copy of the data member you want to capture and then capturing the local copy.

p.223 When a lambda uses an object with static storage duration (defined at global or namespace scope or are declared static inside classes, functions or files), these objects can't be captured by value, even if default by-value capture is specified. Practically speaking, capture by-reference will be employed.

p.223 Default by-value capture is susceptible to dangling pointers (especially "this" pointer), and it misleadingly suggests that lambdas are self-contained.

Item 32 Use init capture to move objects into closures

p.224 Using an init capture makes it possible for you to specify
- the name of a data member in the closure class generated from the lambda.
- an expression initialising that data member.

p.225
```
auto pw = std::make_unique<Widget>();
auto func = [pw = std::move(pw)] { return ...; };
```

To the left of the "=" is the name of the data member in the closure class you're specifying, and to the right is the initialising expression.

p.225 The scope on the left of the "=" is different from the scope on the right. The scope on the left is that of the closure class, whereas the scope on the right is the same as where the lambda is being defined.

p.225 C++14's notion of "capture" is considerably generalised from C++11 because in C++11, it's not possible to capture the result of an expression. As a result, another name for init capture is generalised lambda capture.

p.226 Move capture can be emulated in C++11 by:
- moving the object to be captured into a function object produced by std::bind.
- giving the lambda a reference to the "captured" object.

p.226 C++14 init capture:

```
std::vector<int> data;
auto func = [data = std::move(data)]{...};
```

p.227 C++11 emulation of init capture:

```
std::vector<int> data;
std::bind([](const std::vector<int>& data){...},
          std::move(data));
```

p.227 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. So a std::bind stores its arguments as if passed by value.

p.227 By default, the operator() member function inside a closure generated from a lambda is const. This renders all data members in the closure const within the body of the lambda.

p.228 If the lambda were declared mutable, operator() in its closure class would not be declared const, and it would be appropriate to omit const in the lambda's parameter declaration:

```
std::bind([](std::vector<int>& data) mutable {...},
          std::move(data));
```

Item 33 <u>Use decltype on auto&& parameters to std::forward them.</u>

p.229 C++14 introduced generic lambdas, lambdas that use auto in their parameter specifications.

p.231 We can write a perfect-forwarding lambda like this:

```
auto f = [](auto&& x)
           { return func(std::forward<decltype(x)>(x)); };
```

p.231 C++14 lambdas can also be variadic:

```
auto f = [](auto&&... xs)
           { return func(std::forward<decltype(xs)>(xs)...); };
```

Item 34 <u>Prefer lambdas to std::bind</u>

p.234 std::bind naturally evaluates its arguments when std::bind is called but there may be times when you need evaluation to be deferred until the bind object itself is called, (eg. reading time from clock). Easy to do this with lambdas, more difficult with std::bind.

p.235 When a function being passed to std::bind is overloaded, the function must be cast to the required function pointer type to aid overload resolution which would otherwise fail.

```
std::bind( static_cast<FuncPtrType>(func), ... );
```

p.236 Calling a bind objects call operator may use a function pointer to call the underlying function. (Eg. func is being passed in as a function pointer in the above example). Compilers are less likely to inline function calls through function pointers. It's thus possible that using lambdas generates faster code than using std::bind.

p.238 std::bind always copies its arguments, but callers can achieve the effect of having an argument stored by reference by applying std::ref to it. The result of

```
std::bind(func, std::ref(widget), _1);
```

is that std::bind acts as if it holds a reference to widget.

p.238 All arguments passed to bind objects are passed by reference, because the function call operator for such objects uses perfect forwarding.

p.239 As of C++14, there are no good use cases for std::bind. In C++11, std::bind may be useful for implementing move capture or binding objects with templatised function call operators.

Chapter 7 The Concurrency API

Item 35 Prefer task-based programming to thread-based

p.241

```
int doAsyncWork();
std::thread t(doAsyncWork);                 // thread-based approach
auto fut = std::async(doAsyncWork);         // task-based approach
```

p.245 The std::thread API offers no direct way to get return values from asynchronously run functions, and if those functions throw, the program is terminated.

Thread-based programming calls for manual management of thread exhaustion, oversubscription, load balancing and adaptation to new platforms

Task-based programming via std::async with the default launch policy handles most of these issues for you.

Item 36 <u>Specify std::launch::async if asynchronicity is essential</u>

p.245, 246 Assuming a function f is passed to std::async for execution:
- std::launch::async launch policy means that f must be run asynchronously, ie. on a different thread.
- std::launch::deferred launch policy means that f may run only when get or wait is called on the future returned by std::async. When get or wait is invoked, f will execute synchronously; ie. the caller will block until f finishes running. If neither get nor wait is called, f will never run.

p.246 std::async's default launch policy (when the user doesn't expressly specify one) is "std::launch::async | std::launch::deferred".

p.246 Given a thread t executing: auto fut = std::async(f);
- it's not possible to predict whether f will run concurrently with t.
- it's not possible to predict whether f runs on a thread different from the thread invoking get or wait on fut.
- it may not be possible to predict whether f runs at all.

p. 249 Specify std::launch::async if asynchronous task execution is essential.

```
auto fut = std::async(std::launch::async, f);
```

Item 37 — Make std::threads unjoinable on all paths

p.250 Every std::thread is in one of two states: joinable or unjoinable.

A std::thread that is joinable corresponds to an asynchronous thread of execution that is or could be running, is blocked or waiting to be scheduled, or has run to completion.

p.251 Unjoinable std::thread objects include:
- default-constructed threads that have no function to execute.
- std::thread objects that have been moved from.
- std::threads that have already been joined.
- std::threads that have been detached.

p.251, 253 If a joinable thread's destructor is invoked, execution of the program is terminated. In this manner, the Standardisation Committee decided that the consequences of destroying a joinable thread were sufficiently dire that they essentially banned it.

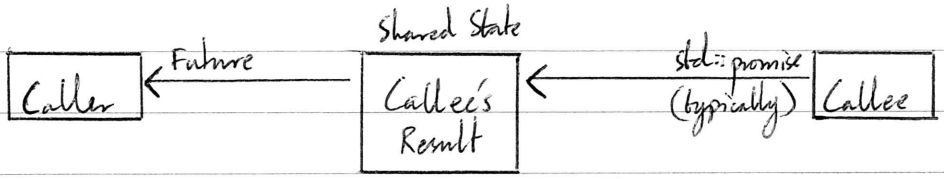
p.255 A std::thread object can change state from joinable to unjoinable only through a member function call, eg. join, detach, or a move operation.

p.253 Employ a "Resource Acquisition Is Initialisation" (RAII) object to manage the thread. In particular, upon its destruction, it may either join or detach its thread as appropriate.

Item 38 — Be aware of varying thread handle destructor behaviour

p.258 Joinable std::threads correspond to underlying system threads of execution. A future for a non-deferred task has a similar relationship to a system thread. As such, both objects can be thought of as handles to system threads.

p.259



p.259 The destructor for the last future referring to a shared state for a non-deferred task launched via std::async blocks until the task completes. In essence, the destructor for such a future does an implicit join.

The destructor for all other futures simply destroys the future object. For asynchronously running tasks, this is akin to an implicit detach. For deferred tasks for which this is the final future, it means that the deferred task will never run.

p.261 Only shared states arising from calls to std::async qualify for the special behaviour. Shared states can be created by other means, one is the use of std::packaged_task.

Item 39 Consider void futures for one-shot event communication

p.262, 266 Suppose a task needs to tell a second, asynchronously running task that a particular event has occurred, because the second task can't proceed until the event has taken place, say. The design is to have the reacting task wait on a future that's set by the detecting task.

p.267 The detecting task can use a std::promise<void> and the reacting task a std::future<void> or std::shared_future<void>. The detecting task will set its std::promise<void> when the event of interest occurs, and the reacting task will wait on its future.

p.268 A std::promise may be set only once. The communications channel between a std::promise and a future is a one-shot mechanism. Condition variables and flag-based designs can be used to communicate multiple times.

Item 40 Use std::atomic for concurrency, volatile for special memory

p.271 Instantiations of the std::atomic template offer operations that are guaranteed to be seen as atomic by other threads. Once a std::atomic object has been constructed, operations on it behave as if they were inside a mutex-protected critical section, but the operations are generally implemented using special machine instructions that are more efficient than would be the case if a mutex were employed.

p.274 Given a sequence of assignments (where a, b, x and y correspond to independent variables), `a = b; x = y;` compilers are permitted to reorder such unrelated assignments. Even if compilers don't reorder them, the underlying hardware might do it (or might make it seem to other cores as if it had), because that can sometimes make the code run faster.

p.274 Std::atomic imposes restrictions on how code can be reordered, and one such restriction is that no code that, in the source code, precedes a write of a std::atomic variable may take place afterwards. (True only for std::atomic using default recommended sequential consistency).

p.276 Volatile is the way we tell compilers that we're dealing with special memory. Its meaning to compilers is "Don't perform any optimisations on operations on this memory."

p.278 std::atomic types do not support copy or move operations. Instead they provide member functions load and store:

```
std::atomic<int> x;
std::atomic<int> y(x.load());   // read x
y.store(x.load());              // read x again
```

Chapter 8 Tweaks

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

p.292 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.

p.292 For lvalue arguments, pass by value (ie. copy construction) followed by move assignment may be significantly more expensive than pass by reference followed by copy assignment. (Due to an extra heap allocation and deallocation).

p.292 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

p.294 emplace_back is available for every standard container that supports push_back. Similarly, every standard container that supports push_front supports emplace_front. And every standard container that supports insert supports emplace.

p.294 Insertion functions take objects to be inserted, while emplacement functions take constructor arguments for objects to be inserted. This difference permits emplacement functions to avoid the creation and destruction of temporary objects that insertion functions can necessitate.

p.295 However, there may be situations where insertion functions run faster. The usual performance-tuning advice thus applies: to determine whether emplacement or insertion runs faster, benchmark them both.

p.295 If all the following are true, emplacement will almost certainly outperform insertion:
- The value being added is constructed into the container, not assigned.
- The argument type(s) being passed differ from the type held by the container.
- The container is unlikely to reject the new value as a duplicate.

p.298 When working with containers of resource-managing objects, you must take care to ensure that if you choose an emplacement function over its insertion counterpart, you're not paying for improved code efficiency with diminished exception safety.

```
std::list<std::shared_ptr<Widget>> ptrs;

void killWidget(Widget* pWidget);          // custom deleter

ptrs.push_back(std::shared_ptr<Widget>(new Widget, killWidget));

ptrs.emplace_back(new Widget, killWidget);   // potential memory leak
```

p.298 When memory allocation involves "new", the returned pointer should be handed over to a resource managing object immediately and in a standalone statement.

p.300 Emplacement functions use direct initialisation, which means they may use explicit constructors. Insertion functions employ copy initialisation, so they can't. Eg. the std::regex constructor taking a const char* pointer is explicit. Hence

```
std::vector<std::regex> regexes;
regexes.emplace_back(nullptr);   // compiles
regexes.push_back(nullptr);      // error!
```