Constructors, Destructors and Assignment Operators

If you don't declare them yourself, compilers will declare their own versions of a copy constructor, a copy assignment operator and a destructor.

Empty e1 → default constructor
Empty e2(e1) → copy constructor
e2 = e1 → copy assignment operator

will not be there if we have added our own.

* By declaring a member function explicitly, you prevent compilers from generating their own version, and by making the function private, you keep people from calling it.
* C++ specifies that when a derived class object is deleted through a pointer to a base class with a non virtual destructor, results are undefined.
  Solution → give the base class a virtual destructor.
* If you are ever tempted to inherit from a standard container or any other class with a non-virtual destructor, resist the temptation.
* Polymorphic base classes should declare virtual destructors.
* If a class has virtual functions, it should have a virtual destructor
* Classes not designed to be base classes or not designed to be used polymorphically should not declare virtual destructors.

Deducing Types

Type deduction in function templates, auto, decltype

The simple idea of template is to pass the data type as a parameter so that we don't need to write the same code for different data types.

```
template <typename T> void f(const T& param);
int x = 0;
f(x);
```

T → deduced to be int
ParamType → deduced to be const int&

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

Case-1: ParamType is a Reference or Pointer but not a Universal Reference

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

Case-2: ParamType is a Universal Reference

(1) If expr = lvalue, both T and ParamType = lvalue
(2) If expr = rvalue, apply Rule-1

Case-3 ParamType is neither a pointer nor a reference

(1) If expr's type is a reference, ignore the reference part.
(2) If expr is const, ignore that too. If it is volatile, ignore that.

* auto type deduction is usually the same as template type deduction, but auto type deduction assumes that a braced initializer represents a std::initializer_list and template type deduction doesn't.
* auto in a function return type or a lambda parameters implies template type deduction, not auto type deduction.

* Given a name or an expression, decltype tells you the name's or the expression's type.
* For lvalue expressions of type T other than names, decltype always reports a type of T&.
* C++14 supports decltype(auto), which like auto deduces a type from its initializer, but it performs the type deduction using the decltype rules.

AUTO

* auto variables have their type deduced from their initializer, so they must be initialized.
* std::function is a template in the C++ STL that generalizes the idea of a function pointer
  std::function can refer to any callable object.
* std::function object typically uses more memory than the auto-declared object.
* std::vector<int> v;
  unsigned sz = v.size();
  official return type = size_type
  On 64 bit windows, unsigned is 32 bits while size_type is 64 bits
  Use auto
* Though std::vector<bool> conceptually holds bools, operator[] for std::vector<bool> doesn't return a reference to an element of the container. Instead, it returns an object of type std::vector<bool>::reference (nested class)
  C++ forbids references to bits.
* Avoid code of the following form:
  auto someVar = expression of "invisible" proxy class type
* The explicitly typed initializer idiom forces auto to declare the type you want it to have.

Modern C++

```
int x(0);
int y = 0;
int z {0};
```
} Three ways of initializing

* Uncopyable objects (std::atomics) may be initialized using braces or parentheses, but not using "="

Braced initialization prohibits implicit narrowing conversions among built-in types. Initialization using parentheses and "=" does not check this.

```
Widget w2 ();   // most vexing parse! declares a function
                // named w2 that returns a Widget!
Widget w3 {};   // calls Widget ctor with no args
```

- Compilers' determination to match braced initializers with constructors taking std::initializer_lists is so strong, it prevails even if the best-match std::initializer_list constructor can't be called.

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

```
std::vector<int> v1 (10,20);  // 10-element vector with each element 20
std::vector<int> v2 {10,20};  // 2-element vector, element values are 10 and 20
```

Nullptr

nullptr's advantage is that it does not have an integral type, it is a pointer of all types.

Its actual type is nullptr_t.

Enums

* C++98 style enums = unscoped enums
* Enumerators of scoped enums are visible only within the enum. They convert to other types only with a cast.
* Both scoped enums 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. Unscoped enums may be forward-declared only if their declaration specifies an underlying type.

Prefer deleted functions to private undefined ones.
Any function may be deleted, including non-member functions and template instantiations.

override

For overriding to occur, several requirements must be met:

- Base function must be virtual
- Base and derived function names must be identical
- Parameter types identical
- Constness identical
- Return types compatible
- Reference qualifiers identical.

* C++ makes it explicit that a derived class function is supposed to override a base class version: declare it override.

Applying final to a virtual function prevents the function from being overridden in derived classes. final may also be supplied to a class, in which case the class is prohibited from being used as a base class.

Declare overriding functions override

Member functions reference qualifiers make it possible to treat lvalue and rvalue objects (*this) differently.

Prefer const_iterators to iterators

const_iterators are the STL equivalent of pointers-to-const

Prefer const_iterators to iterators.

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

Declare functions noexcept if they won't emit exceptions

In C++11, unconditional noexcept is for functions that guarantee they won't emit exceptions.

```
int f(int x) noexcept;
```

In C++11, by default all memory deallocation functions and all destructors — both user-defined and compiler generated — are implicitly noexcept.

noexcept functions are more optimizable than non-noexcept functions.

noexcept is particularly valuable for the move operations, swap, memory deallocation functions, and destructors.

Constexpr

Conceptually, constexpr indicates a value that is not only constant, it is known during compilation.

Constexpr function:

(1) if the arguments are known during compilation, the result will be computed during compilation

(2) when called with one or more values that are not known during compilation, it acts like a normal function, computing its result at runtime.

Make const member functions thread safe unless you are certain they'll never be used in a concurrent context.

Use of std::atomic variables may offer better performance than a mutex, but they're suited for manipulation of only a single variable or memory location

Special Member Functions

C++98 has four such functions — the default constructor, the destructor, the copy constructor and the copy assignment operator

C++11 two more — the move constructor, move assignment operator

A move constructor prevents a move assignment operator from being generated, and declaring a move assignment operator prevents compilers from generating a move constructor

Move operations won't be generated for any class that explicitly declares a copy operation. (and vice versa)

The Rule of Three states that if you declare any of a copy constructor, copy assignment operator or destructor, you should declare all three.

Member function templates never suppress generation of special member functions.

11 Dec, 2024

Smart Pointers

There are 4 smart pointers in C++11 : auto_ptr, unique_ptr, shared_ptr, weak_ptr.

std::unique_ptr - small, fast, move only for managing resources with exclusive ownership semantics resource destruction via delete, but custom deleters can be specified. stateful deleters and function pointers as deleters increase the size of unique_ptr objects. Converting to shared_ptr is easy.

for std::unique_ptr, the type of the deleter is part of the type of the smart pointer. For std::shared_ptr, it is not.

shared_ptr's reference count is part of a larger data structure called control block.

make_shared always creates a control block → not from shared_ptr and weak_ptr

Also from unique_ptr, auto_ptr and raw pointer

std::enable_shared_from_this defines a member function that creates a shared_ptr to the current object, but it does without duplicating control blocks.

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

weak_ptrs that dangle are said to have expired.

```
if (wpw.expired())
```

weak_ptr::lock returns shared_ptr

Potential uses cases for weak_ptrs include caching, observer lists and the prevention of std::shared_ptr cycles.

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

Make functions inappropriate → custom deleters
pass braced initializers

For shared_ptrs: (1) classes with custom memory management
(2) systems with memory concerns

Pimpl → pointer to implementation
Decreases build times by reducing compilation dependencies between class clients and class implementations.
For std::unique_ptr pImpl pointers declare special member functions in the class header, but implement them in the implementation file.
(does not apply to shared_ptrs)

R-value references, move semantics, perfect forwarding

std::move casts its argument to an rvalue.
Don't declare objects const if you want to be able to move from them.
std::move doesn't guarantee that the object it's casting will be eligible to be moved.
std::forward is a conditional cast.
→ it casts to an rvalue only if its argument was initialized with an rvalue.

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 are initialized with rvalues. They correspond to lvalue references if they are initialized with lvalues.

rvalue references should be unconditionally cast to rvalues (via std::move) and universal references should be conditionally cast to rvalues (via std::forward) (the last time they are used)

Never apply std::move or std::forward to local objects if they would otherwise be eligible for the return value optimization.

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 are typically better matches than copy constructors for non-const lvalues, and they can hijack derived class calls to base class copy and move constructors.

Alternatives to the combination of universal references and overloading include the use of distinct function names, passing parameters by lvalue-reference-to-const, passing parameters by value, and using tag-dispatch.

Constraining templates via std::enable_if permits the use of universal references and overloading together, but it controls the conditions under which compilers may use the universal reference overloads.

```
template<typename T>
void func(T&& param);

Widget w;
func(w);                    // call func with lvalue; T deduced to be Widget&.
func(widgetFactory());      // call func with rvalue; T deduced to be Widget.
```

We are forbidden from declaring references to references
Compilers may produce them in template instantiation.

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.

→ reference collapsing

Reference collapsing occurs in 4 contexts: template instantiation, auto type generation, creation and use of typedefs and alias declarations and decltype.

There are several scenarios where C++11 move semantics do you no good:
(1) No move operations (object does not offer)
(2) Move not faster
(3) Move not usable (function not declared noexcept)
(4) Source object is lvalue.

The use of a braced initializer is a perfect forwarding failure case.

Neither 0 nor NULL can be perfect forwarded as a null pointer.

The kind 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.

Lambda Expressions

Default by-reference capture can lead to dangling references.

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

Use C++14's init capture to move objects into closures.

In C++11, emulate init capture via hand-written classes or std::bind.

Use decltype on auto&& parameters to std::forward them.

Lambdas are more readable, more expressive, and may be more efficient than using std::bind. In C++11 only, std::bind may be useful for implementing move capture or for binding objects with templatized call operators.

Friend something to allow your class to grant access to another class or function

Friendship isn't inherited, transitive or reciprocal