Effective C++

Accustoming Yourself to C++

1. View C++ as a federation of languages
2. Prefer consts, enums, and inlines to #defines
3. Use const whenever possible
4. Make sure that objects are initialized before they're used

Constructors, Destructors, and Assignment Operators

1. Know what functions C++ silently writes and calls
2. Explicitly disallow use of compiler-generated functions you do not want
3. Declare destructors virtual in polymorphic base classes
4. Prevent exceptions from leaving destructors
5. Never call virtual functions during construction or destruction
6. Have assignment operators return a reference to *this
7. Handle assignment to self in operator=
8. Copy all parts of an object

Resource Management

1. Use objects to manage resources (RAII)
2. Think carefully about copying behavior in resource managing classes
3. Provide access to raw resources in resource-managing classes
4. Use the same form in corresponding uses of new and delete
5. Store newed objects in smart pointers in standalone statements

Designs and Declarations

1. Make interfaces easy to use correctly and hard to use incorrectly
2. Treat class design as type design
3. Prefer pass-by-reference-to-const to pass-by-value
4. Don't try to return a reference when you must return an object
5. Declare data members private
6. Prefer non-member non-friend functions to member functions
7. Declare non-member functions when type conversions should apply to all parameters
8. Consider support for a non-throwing swap

Implementations

1. Postpone variable definitions as long as possible
2. Minimize casting
3. Avoid returning "handles" to object internals
4. Strive for exception-safe code
5. Understand the ins and outs of inlining
6. Minimize compilation dependencies between files

Inheritance and Object-Oriented Design

1. Make sure public inheritance models "is-a"
2. Avoid hiding inherited names
3. Differentiate between inheritance of interface and inheritance of implementation
4. Consider alternatives to virtual functions
5. Never redefine an inherited non-virtual function
6. Never redefine a function's inherited default parameter value
7. Model "has-a" or "is-implemented-in-terms-of" through composition
8. Use private inheritance judiciously
9. Use multiple inheritance judiciously

Templates and Generic Programming

1. Understand implicit interfaces and compile-time polymorphism
2. Understand the two meanings of typename
3. Know how to access names in templatized base classes
4. Factor parameter-independent code out of templates
5. Use member function templates to accept "all compatible types"
6. Define non-member functions inside templates when type conversions are desired
7. Use traits classes for information about types
8. Be aware of template metaprogramming

Customizing new and delete

1. Understand the behaviour of the new-handler
2. Understand when it makes sense to replace new and delete
3. Adhere to convention when writing new and delete

4. Write placement delete if you write placement new

Miscellany

1. Pay attention to compiler warnings
2. Familiarize yourself with the standard library
3. Familiarize yourself with Boost

More Effective C++

Basics

1. Distinguish between pointers and references
2. Prefer C++ style casts
3. Never treat arrays polymorphically
4. Avoid gratuitous default constructors

Operators

1. Be wary of user-defined conversion functions
2. Distinguish between prefix and postfix forms of increment and decrement operators
3. Never overload &&, ll, or ,
4. Understand the different meanings of new and delete
    - new operator - the language keyword/operator that is used in code
    - operator new - the underlying function called by the new operator
    - placement new - `new(location) T` constructs an object of type, T, in memory location
        - NOTE: this functionality is encapsulated by Allocators

Exceptions

1. Use destructors to prevent resource leaks
2. Prevent resource leaks in constructors
3. Prevent exceptions from leaving destructors
4. Understand how throwing an exception differs from passing a parameter or calling a virtual function
5. Catch exceptions by reference
6. Use exception specifications judiciously

7. Understand the costs of exception handling

Efficiency

1. Remember the 80-20 (90-10) rule
2. Consider using lazy evaluation
3. Amortize the cost of expected computations
4. Understand the origin of temporary objects
5. Facilitate the return value optimization
6. Overload to avoid implicit type conversions
7. Consider using op= instead of stand-alone op
8. Consider alternative libraries
9. Understand the costs of virtual functions, multiple inheritance, virtual base classes, and RTTI

Techniques

1. Virtualizing constructors and non-member functions
2. Limiting the number of objects of a class
3. Requiring or prohibiting heap-based objects
4. Smart pointers
5. Reference counting
6. Proxy classes
7. Making functions virtual with respect to more than one object

Miscellany

1. Program in the future tense
2. Make non-leaf classes abstract
3. Understand how to combine C++ and C in the same program
4. Familiarize yourself with the language standard

Effective STL

Containers

1. Choose your containers with care
2. Beware the illusion of container independent code
3. Make copying cheap and correct for objects in containers
4. Call empty instead of checking size() against zero

5. Prefer range member functions to their single-element counterparts
6. Be alert for C++'s most vexing parse
7. When using containers of newed pointers, remember to delete the pointers when the container is destroyed
8. Never create containers of auto_pts
9. Choose carefully among erasing options
10. Be aware of allocator conventions and restrictions
11. Understand the legitimate uses of custom allocators
12. Have realistic expectations about the thread safety of STL containers

vector and string

1. Prefer vector and string to dynamically allocated arrays
2. Use reserve to avoid unnecessary reallocations
3. Be aware of variations in string implementations (some removed by C++11)
4. Know how to pass vector and string data to legacy APIs
5. Use "the swap trick" to trim excess capacity (use .shrink_to_fit after C++11)
6. Avoid using vector

Associative Containers

1. Understand the difference between equality and equivalence
2. Specify comparison types for associative containers
3. Always have comparison functions return false for equal values
4. Avoid in-place key modification in set and multiset
5. Consider replacing associative containers with sorted vectors
6. Choose carefully between `map::operator[]` and map::insert when efficiency is important
7. Familiarize yourself with the nonstandard hashed containers

Iterators

1. Prefer iterator to const_iterator, reverse_iterator, and const_reverse_iterator
2. Use distance and advance to convert a container's const_iterators to iterators
3. Understand how to use reverse iterator's base iterator
4. Consider istreambuf iterators for character-by-character input

Algorithms

1. Make sure destination ranges are big enough
2. Know your sorting options
3. Follow remove-like algorithms by erase if you really want to remove something

4. Be wary of remove-like algorithms on containers of pointers
5. Note which algorithms expect sorted ranges
6. Implement simple case-insensitive string comparisons via mismatch or lexicographical_compare
7. Understand the proper implementation of copy_if
8. Use accumulate or for_each to summarize ranges

Functors, Functor Classes, Functions, etc

1. Design functor classes for pass-by-value
2. Make predicates pure functions
3. Make functor classes adaptable
4. Understand the reasons for ptr_fun, mem_fun, and mem_fun_ref
5. Make sure less means operator<

Programming with the STL

1. Prefer algorithm calls to hand-written loops
2. Prefer member functions to algorithms with the same names
3. Distinguish among count, find, binary_search, lower_bound, upper_bound, and equal_range
4. Consider function objects instead of functions as algorithm parameters
5. Avoid producing write-only code
6. Always #include the proper headers
7. Learn to decipher STL-related compiler diagnostics
8. Familiarize yourself with STL-related web sites

Effective Modern C++

Deducing Types

1. Understand template type deduction
2. Understand auto type deduction
3. Understand decltype
4. Know how to view deduced types

auto

1. Prefer auto to explicit type declarations

2. Use the explicitly typed initializer idiom when auto deduces undesired types

Moving to Modern C++

1. Distinguish between () and {} when creating objects
2. Prefer nullptr to 0 and NULL
3. Prefer alias declarations to typedefs
4. Prefer scoped enums to unscoped enums
5. Prefer deleted functions to private undefined ones
6. Declare overriding functions override
7. Prefer const_iterators to iterators
8. Declare functions noexcept if they won't emit exceptions
9. Use constexpr whenever possible
10. Make const member functions thread safe
11. Understand special member function generation

Smart Pointers

1. Use std::unique_ptr for exclusive-ownership resource management
2. Use std::shared_ptr for shared-ownership resource management
3. Use std::weak_ptr for std::shared_ptr-like pointers that can dangle
4. Prefer std::make_unique and std::make_shared to direct use of new
5. When using the Pimpl idiom, define special member functions in the implementation file

Rvalue References, Move Semantics, and Perfect Forwarding

1. Understand std::move and std::forward
2. Distinguish universal references from rvalue references
3. Use std::move on rvalue references, std::forward on universal references
4. Avoid overloading on universal references
5. Familiarize yourself with overloading on universal references
6. Understand reference collapsing
7. Assume that move operations are not present, are not cheap, and not used
8. Familiarize yourself with perfect forwarding failure cases

Lambda Expressions

1. Avoid default capture modes
2. Use init capture to move objects into closures
3. Use decltype on auto&& parameters to std::forward them

4. Prefer lambdas to std::bind

The Concurrency API

1. Prefer task-based programming to thread-based
2. Specify std::launch::async if asynchronicity is essential
3. Make std::threads unjoinable on all paths
4. Be aware of varying thread handle destructor behavior
5. Consider void futures for one-shot event communication
6. Use std::atomic for concurrency, volatile for special memory

Tweaks

1. Consider pass-by-value for copyable parameters that are cheap to move and always copied
2. Consider emplacement instead of insertion