Chapter 2 - The Basics

1. Don’t panic! All will become clear in time
2. You don’t have to know every detail of C++ to write good programs
3. Focus on programming techniques, not on language features

Chapter 3 - Abstraction Mechanisms

1. Express ideas directly in code
2. Define classes to represent application concepts directly in code
3. Use concrete classes to represent simple concepts and performance-critical components
4. Avoid "naked" new and delete operations
5. Use resource handles and RAII to manage resources
6. Use abstract classes as interfaces when complete separation of interface and implementation is needed
7. Use class hierarchies to represent concepts with inherent hierarchical structure
8. When designing a class hierarchy, distinguish between implementation inheritance and interface inheritance
9. Control construction, copy, move, and destruction of objects
10. Return containers by value (relying on move for efficiency)
11. Provide strong resource safety
12. Use containers, defined as resource handle templates, to hold collections of values of the same type
13. Use function templates to represent general algorithms
14. Use function objects, including lambdas, to represent policies and actions
15. Use type and template aliases to provide a uniform notation for types that may vary among similar types or among implementations

Chapter 4 - Containers and Algorithms

1. Don’t reinvent the wheel
2. When you have a choice, prefer the standard library over other libraries
3. Do not think that the standard library is ideal for everything
4. Remember to #include the headers for the facilities you use
5. Remember that standard-library facilities are defined in namespace std
6. Prefer strings over C-style strings

7. iostreams are type sensitive, type-safe, and extensible
8. Prefer vector, map<K,T>, and unordered_map<K,T> over `T[]`
9. Know your standard containers and their tradeoffs
10. Use vector as your default container
11. Prefer compact data structures
12. If in doubt, use a range-checked vector (such as Vec)
13. Use push_back() or back_inserter() to add elements to a container
14. Use push_back() on a vector rather than realloc() on an array
15. Catch common exceptions in main()
16. Know your standard algorithms and prefer them over handwritten loops
17. If iterator use gets tedious, define container algorithms

Chapter 5 - Concurrency and Utilities

1. Use resource handles to manage resources (RAII)
2. Use unique_ptr to refer to objects of polymorphic type
3. Use shared_ptr to refer to shared objects
4. Use type-safe mechanisms for concurrency
5. Minimize the use of shared data
6. Don't choose shared data for communication because of "efficiency" without thought and preferably not without measurement
7. Think in terms of concurrent tasks, rather than threads
8. A library doesn't have to be large or complicated to be useful
9. Time your programs before making claims about efficiency
10. You can write code to explicitly depend on properties of types
11. Use regular expressions for simple pattern matching
12. Don't try to do serious numeric computation using only the language
13. Properties of numeric types are accessible through numeric_limits

Chapter 6 - Types and Declarations

1. For the final word on language definition issues, see the ISO C++ standard
2. Avoid unspecified and undefined behavior
3. Isolate code that must depend on implementation-defined behavior
4. Avoid unnecessary assumptions about the numeric value of characters
5. Remember that an integer starting with a 0 is octal
6. Avoid "magic constants"
7. Avoid unnecessary assumptions about the size of integers

8. Avoid unnecessary assumptions about the range and precision of floating-point types
9. Prefer plain char over signed char and unsigned char
10. Beware of conversions between signed and unsigned types
11. Declare one name (only) per declaration
12. Keep common and local names short, and keep uncommon and non-local names longer
13. Avoid similar-looking names
14. Name an object to reflect its meaning rather than its type
15. Maintain a consistent naming style
16. Avoid ALL_CAPS names
17. Keep scopes small
18. Don’t use the same name in both a scope and an enclosing scope
19. Prefer the {}-initializer syntax for declarations with a named type
20. Prefer the = syntax for the initialization in declarations using auto
21. Avoid uninitialized variables
22. Use an alias to define a meaningful name for a built-in type in cases in which the built-in type used to represent a value might change
23. Use an alias to define synonyms for types

Chapter 7 - Pointers, Arrays, and References

1. Keep use of pointers simple and straightforward
2. Avoid nontrivial pointer arithmetic
3. Take care not to write beyond the bounds of an array
4. Avoid multidimensional arrays
5. Use nullptr rather than 0 or NULL
6. Use containers (e.g., vector, array, and valarray) rather than built-in (C-style) arrays
7. Use string rather than zero-terminated arrays of char
8. Use raw strings for string literals with complicated uses of backslash
9. Prefer const reference arguments to plain reference arguments
10. Use rvalue references (only) for forwarding and move semantics
11. Keep pointers that represent ownership inside handle classes
12. Avoid void* except in low-level code
13. Use const pointers and const references to express immutability in interfaces
14. Prefer references to pointers as arguments, except where "no object" is a reasonable option

Chapter 8 - Structures, Unions, and Enumerations

1. When compactness of data is important, lay out structure data members with larger members before smaller ones
2. Use bit-fields to represent hardware-imposed data layouts
3. Don’t naively try to optimize memory consumption by packing several values into a single byte
4. Use unions to save space (represent alternatives) and never for type conversion
5. Use enumerations to represent sets of named constants
6. Prefer class enums over "plain" enums to minimize surprises
7. Define operations on enumerations for safe and simple use

Chapter 9 - Statements

1. Don’t declare a variable until you have a value to initialize it with
2. Prefer a switch-statement to an if-statement when there is a choice
3. Prefer a range-for-statement to a for-statement when there is a choice
4. Prefer a for-statement to a while-statement when there is an obvious loop variable
5. Prefer a while-statement to a for-statement when there is no obvious loop variable
6. Avoid do-statements
7. Avoid goto
8. Keep comments crisp
9. Don’t say in comments what can be clearly stated in code
10. State intent in comments
11. Maintain a consistent indentation style

Chapter 10 - Expressions

1. Prefer the standard library to other libraries and to "handcrafted code"
2. Use character-level input only when you have to
3. When reading, always consider ill-formed input
4. Prefer suitable abstractions (classes, algorithms, etc.) to direct use of language features (e.g. ints, statements)
5. Avoid complicated expressions
6. If in doubt about operator precedence, parenthesize
7. Avoid expressions with undefined order of evaluation

8. Avoid narrowing conversions
9. Define symbolic constants to avoid "magic constants"
10. Avoid narrowing conversions

Chapter 11 - Select Operations

1. Prefer prefix ++ over suffix ++
2. Use resource handles to avoid leaks, premature deletion, and double deletion
3. Don’t put objects on the free store if you don’t have to
4. Avoid "naked new" and "naked delete"
5. Use RAII
6. Prefer a named function object to a lambda if the operation requires comments
7. Prefer a named function object to a lambda if the operation is generally useful
8. Keep lambdas short
9. For maintainability and correctness, be careful about capture by reference
10. Let the compiler deduce the return type of a lambda
11. Use the T{e} notation for construction
12. Avoid explicit type conversion (casts)
13. When explicit type conversion is necessary, prefer a named cast
14. Consider using a run-time checked cast, such as narrow_cast<>(), for conversion between numeric types

Chapter 12 - Functions

1. "Package" meaningful operations as carefully named functions
2. A function should perform a single logical operation
3. Keep functions short
4. Don’t return pointers or references to local variables
5. If a function may have to be evaluated at compile time, declare it constexpr
6. If a function cannot return, mark it `[[noreturn]]`
7. Use pass-by-value for small objects
8. Use pass-by-const-reference to pass large values that you don’t need to modify
9. Return a result as a return value rather than modifying an object through an argument
10. Use rvalue references to implement move and forwarding
11. Pass a pointer if "no object" is a valid alternative (and represent "no object" by nullptr)
12. Use pass-by-non-const-reference only if you have to
13. Use const extensively and consistently
14. Assume that a char* or a const char* argument points to a C-style string

15. Avoid passing arrays as pointers
16. Pass a homogeneous list of unknown length as an initializer_list (or as some other container)
17. Avoid unspecified numbers of arguments (...)
18. Use overloading when functions perform conceptually the same task on different types
19. When overloading on integers, provide functions to eliminate common ambiguities
20. Specify preconditions and postconditions for your functions
21. Prefer function objects (including lambdas) and virtual functions to pointers to functions
22. Avoid macros
23. If you must use macros, use ugly names with lots of capital letters

Chapter 13 - Exception Handling

1. Develop an error-handling strategy early in a design
2. Throw an exception to indicate that you cannot perform an assigned task
3. Use exceptions for error handling
4. Use purpose-designed user-defined types as exceptions (not built-in types)
5. If you for some reason cannot use exceptions, mimic them
6. Use hierarchical error handling
7. Keep the individual parts of error handling simple
8. Don’t try to catch every exception in every function
9. Always provide the basic guarantee
10. Provide the strong guarantee unless there is a reason not to
11. Let a constructor establish an invariant, and throw if it cannot
12. Release locally owned resources before throwing an exception
13. Be sure that every resource acquired in a constructor is released when throwing an exception in that constructor
14. Don’t use exceptions where more local control structures will suffice
15. Use the "Resource Acquisition Is Initialization" technique to manage resources
16. Minimize the use of try-blocks
17. Not every program needs to be exception-safe
18. Use "Resource Acquisition Is Initialization" and exception handlers to maintain invariants
19. Prefer proper resource handles to the less structured finally
20. Design your error-handling strategy around invariants
21. What can be checked at compile time is usually best checked at compile time (using static_assert)
22. Design your error-handling strategy to allow for different levels of checking/enforcement
23. If your function may not throw, declare it noexcept

24. Don’t use exception specification
25. Catch exceptions that may be part of a hierarchy by reference
26. Don’t assume that every exception is derived from class exception
27. Have main() catch and report all exceptions
28. Don’t destroy information before you have its replacement ready
29. Leave operands in valid states before throwing an exception from an assignment
30. Never let an exception escape from a destructor
31. Keep ordinary code and error-handling code separate
32. Beware of memory leaks caused by memory allocated by new not being released in case of an exception
33. Assume that every exception that can be thrown by a function will be thrown
34. A library shouldn’t unilaterally terminate a program. Instead, throw an exception and let a caller decide
35. A library shouldn’t produce diagnostic output aimed at an end user. Instead, throw an exception and let a caller decide

Chapter 14 - Namespaces

1. Use namespaces to express logical structure
2. Place every nonlocal name, except main(), in some namespace
3. Design a namespace so that you can conveniently use it without accidentally gaining access to unrelated namespaces
4. Avoid very short names for namespaces
5. If necessary, use namespace aliases to abbreviate long namespace names
6. Avoid placing heavy notational burdens on users of your namespaces
7. Use separate namespaces for interfaces and implementations
8. Use the Namespace::member notation when defining namespace members
9. Use inline namespaces to support versioning
10. Use using-directives for transition, for foundational libraries (such as std), or within a local scope
11. Don’t put a using-directive in a header file

Chapter 15 - Source Files and Programs

1. Use header files to represent interfaces and to emphasize logical structure
2. #include a header in the source file that implements its functions
3. Don’t define global entities with the same name and similar-but-different meanings in different translation units
4. Avoid non-inline function definitions in headers

5. Use #include only at global scope and in namespaces
6. #include only complete declarations
7. Use include guards
8. #include C headers in namespaces to avoid global names
9. Make headers self-contained
10. Distinguish between users' interfaces and implementers' interfaces
11. Distinguish between average users' interfaces and expert users' interfaces
12. Avoid non-local objects that require run-time initialization in code intended for use as part of non-C++ programs

Chapter 16 - Classes

1. Represent concepts as classes
2. Separate the interface of a class from its implementation
3. Use public data (structs) only when it really is just data and no invariant is meaningful for the data members
4. Define a constructor to handle initialization of objects
5. By default declare single-argument constructors explicit
6. Declare a member function that does not modify the state of its object const
7. A concrete type is the simplest kind of class. Where applicable, prefer a concrete type over more complicated classes and over plain data structures
8. Make a function a member only if it needs direct access to the representation of a class
9. Use a namespace to make the association between a class and its helper functions explicit
10. Make a member function that doesn't modify the value of its object a const member function
11. Make a function that needs access to the representation of a class but needn't be called for a specific object a static member function

Chapter 17 - Construction, Cleanup, Copy, and Move

1. Design constructors, assignments, and the destructor as a matched set of operations
2. Use a constructor to establish an invariant for a class
3. If a constructor acquires a resource, its class needs a destructor to release the resource
4. If a class has a virtual function, it needs a virtual destructor
5. If a class does not have a constructor, it can be initialized by memberwise initialization
6. Prefer {} initialization over = and () initialization
7. Give a class a default constructor if and only if there is a "natural" default value
8. If a class is a container, give it an initializer-list constructor

9. Initialize members and bases in their order of declaration
10. If a class has a reference member, it probably needs copy operations (copy constructor and copy assignment)
11. Prefer member initialization over assignment in a constructor
12. Use in-class initializers to provide default values
13. If a class is a resource handle, it probably needs copy and move operations
14. When writing a copy constructor, be careful to copy every element that needs to be copied (beware of default initializers)
15. A copy operations should provide equivalence and independence
16. Beware of entangled data structures
17. Prefer move semantics and copy-on-write to shallow copy
18. If a class is used as a base class, protect against slicing
19. If a class needs a copy operation or a destructor, it probably needs a constructor, a destructor, a copy assignment, and a copy constructor
20. If a class has a pointer member, it probably needs a destructor and non-default copy operations
21. If a class is a resource handle, it needs a constructor, a destructor, and non-default copy operations
22. If a default constructor, assignment, or destructor is appropriate, let the compiler generate it (don't rewrite it yourself)
23. Be explicit about your invariants maintain them
24. Make sure that copy assignments are safe for self-assignment
25. When adding a new member to a class, check to see if there are user-defined constructors that need to be updated to initialize the member

Chapter 18 - Overloading

1. Define operators primarily to mimic conventional usage
2. Redefine or prohibit copying if the default is not appropriate for a type
3. For large operands, use const reference argument types
4. For large results, use a move constructor
5. Prefer member functions over nonmembers for operations that need access to the representa- tion
6. Prefer nonmember functions over members for operations that do not need access to the rep- resentation
7. Use namespaces to associate helper functions with "their" class
8. Use nonmember functions for symmetric operators
9. Use member functions to express operators that require an lvalue as their left-hand operand
10. Use user-defined literals to mimic conventional notation

11. Provide "set() and get() functions" for a data member only if the fundamental semantics of a class require them
12. Be cautious about introducing implicit conversions
13. Avoid value-destroying ("narrowing") conversions
14. Do not define the same conversion as both a constructor and a conversion operator

Chapter 19 - Special Operators

1. Use operator.() for subscripting and for selection based on a single value
2. Use operator()() for call semantics, for subscripting, and for selection based on multiple values
3. Use operator–>() to dereference "smart pointers"
4. Prefer prefix ++ over suffix ++
5. Define the global operator new() and operator delete() only if you really have to
6. Define member operator new() and member operator delete() to control allocation and deallocation of objects of a specific class or hierarchy of classes
7. Use user-defined literals to mimic conventional notation
8. Place literal operators in separate namespaces to allow selective use
9. For non-specialized uses, prefer the standard string to the result of your own exercises
10. Use a friend function if you need a nonmember function to have access to the representation of a class (e.g., to improve notation or to access the representation of two classes)
11. Prefer member functions to friend functions for granting access to the implementation of a class

Chapter 20 - Derived Classes

1. Avoid type fields
2. Access polymorphic objects through pointers and references
3. Use abstract classes to focus design on the provision of clean interfaces
4. Use override to make overriding explicit in large class hierarchies
5. Use final only sparingly
6. Use abstract classes to specify interfaces
7. Use abstract classes to keep implementation details out of interfaces
8. A class with a virtual function should have a virtual destructor
9. An abstract class typically doesn’t need a constructor
10. Prefer private members for implementation details
11. Prefer public members for interfaces
12. Use protected members only carefully when really needed
13. Don’t declare data members protected

Chapter 21 - Class Hierarchies

1. Use unique_ptr or shared_ptr to avoid forgetting to delete objects created using new
2. Avoid data members in base classes intended as interfaces
3. Use abstract classes to express interfaces
4. Give an abstract class a virtual destructor to ensure proper cleanup
5. Use override to make overriding explicit in large class hierarchies
6. Use abstract classes to support interface inheritance
7. Use base classes with data members to support implementation inheritance
8. Use ordinary multiple inheritance to express a union of features
9. Use multiple inheritance to separate implementation from interface
10. Use a virtual base to represent something common to some, but not all, classes in a hierarchy

Chapter 22 - Run-Time Type Information

1. Use virtual functions to ensure that the same operation is performed independently of which interface is used for an object
2. Use dynamic_cast where class hierarchy navigation is unavoidable
3. Use dynamic_cast for type-safe explicit navigation of a class hierarchy
4. Use dynamic_cast to a reference type when failure to find the required class is considered a failure
5. Use dynamic_cast to a pointer type when failure to find the required class is considered a valid alternative
6. Use double dispatch or the visitor pattern to express operations on two dynamic types (unless you need an optimized lookup)
7. Don’t call virtual functions during construction or destruction
8. Use typeid to implement extended type information
9. Use typeid to find the type of an object (and not to find an interface to an object)
10. Prefer virtual functions to repeated switch-statements based on typeid or dynamic_cast

Chapter 23 - Templates

1. Use templates to express algorithms that apply to many argument types
2. Use templates to express containers
3. Note that template and template are synonymous
4. When defining a template, first design and debug a non-template version adding parameters
5. Templates are type-safe, but checking happens too late

6. When designing a template, carefully consider the concepts (requirements) assumed for its template arguments
7. If a class template should be copyable, give it a non-template copy constructor and a non-template copy assignment
8. If a class template should be movable, give it a non-template move constructor and a non-template move assignment
9. A virtual function member cannot be a template member function
10. Define a type as a member of a template only if it depends on all the class template's arguments
11. Use function templates to deduce class template argument types
12. Overload function templates to get the same semantics for a variety of argument types
13. Use argument substitution failure to provide just the right set of functions for a program
14. Use template aliases to simplify notation and hide implementation details
15. There is no separate compilation of templates: #include template definitions in every translation unit that uses them
16. Use ordinary functions as interfaces to code that cannot deal with templates
17. Separately compile large templates and templates with nontrivial context dependencies

Chapter 24 - Generic Programming

1. A template can pass argument types without loss of information
2. Templates provide a general mechanism for compile-time programming
3. Templates provide compile-time "duck typing"
4. Design generic algorithms by "lifting" from concrete examples
5. Generalize algorithms by specifying template argument requirements in terms of concepts
6. Do not give unconventional meaning to conventional notation
7. Use concepts as a design tool
8. Aim for "plug compatibility" among algorithms and argument type by using common and regular template argument requirements
9. Discover a concept by minimizing an algorithm's requirements on its template arguments and then generalizing for wider use
10. A concept is not just a description of the needs of a particular implementation of an algorithm
11. If possible, choose a concept from a list of well-known concepts
12. The default concept for a template argument is Regular
13. Not all template argument types are Regular
14. A concept requires a semantic aspect
15. Make concepts concrete in code
16. Express concepts as compile-time predicates (constexpr functions) and test them using static_assert() or enable_if<>

17. Use axioms as a design tool
18. Use axioms as a guide for testing
19. Some concepts involve two or more template arguments
20. Concepts are not just types of types
21. Concepts can involve numeric values
22. Use concepts as a guide for testing template definitions

Chapter 25 - Specialization

1. Use templates to improve type safety
2. Use templates to raise the level of abstraction of code
3. Use templates to provide flexible and efficient parameterization of types and algorithms
4. Remember that value template arguments must be compile-time constants
5. Use function objects as type arguments to parameterize types and algorithms with "policies"
6. Use default template arguments to provide simple notation for simple uses
7. Specialize templates for irregular types (such as arrays)
8. Specialize templates to optimize for important cases
9. Define the primary template before any specialization
10. A specialization must be in scope for every use

Chapter 26 - Instantiation

1. Let the compiler/implementation generate specializations as needed
2. Explicitly instantiate if you need exact control of the instantiation environment
3. Explicitly instantiate if you optimize the time needed to generate specializations
4. Avoid subtle context dependencies in a template definition
5. Names must be in scope when used in a template definition or findable through argument-dependent lookup (ADL)
6. Keep the binding context unchanged between instantiation points
7. Avoid fully general templates that can be found by ADL
8. Use concepts and/or static_assert to avoid using inappropriate templates
9. Use using-declarations to limit the reach of ADL
10. Qualify names from a template base class with –> or T:: as appropriate

Chapter 27 - Templates and Hierarchies

1. When having to express a general idea in code, consider whether to represent it as a template or as a class hierachy

2. A template usually provides common code for a variety of arguments
3. An abstract class can completely hide implementation details from users
4. Irregular implementations are usually best represented as derived classes
5. If explicit use of free store is undesirable, templates have an advantage over class hierarchies
6. Templates have an advantage over abstract classes where inlining is important
7. Template interfaces are easily expressed in terms of template argument types
8. If run-time resolution is needed, class hierarchies are necessary
9. The combination of templates and class hierarchies is often superior to either without the other
10. Think of templates as type generators (and function generators)
11. There is no default relation between two classes generated from the same template
12. Do not mix class hierarchies and arrays
13. Do not naively templatize large class hierarchies
14. A template can be used to provide a type-safe interface to a single (weakly typed) implementation
15. Templates can be used to compose type-safe and compact data structures
16. Templates can be used to linearize a class hierarchy (minimizing space and access time)

Chapter 28 - Metaprogramming

1. Use metaprogramming to improve type safety
2. Use metaprogramming to improve performance by moving computation to compile time
3. Avoid using metaprogramming to an extent where it significantly slows down compilation
4. Think in terms of compile-time evaluation and type functions
5. Use template aliases as the interfaces to type functions returning types
6. Use constexpr functions as the interfaces to type functions returning (non-type) values
7. Use traits to non-intrusively associate properties with types
8. Use Conditional to choose between two types
9. Use Select to choose among several alternative types
10. Use recursion to express compile-time iteration
11. Use metaprogramming for tasks that cannot be done well at run time
12. Use Enable_if to selectively declare function templates
13. Concepts are among the most useful predicates to use with Enable_if
14. Use variadic templates when you need a function that takes a variable number of arguments of a variety of types
15. Don’t use variadic templates for homogeneous argument lists (prefer initializer lists for that)
16. Use variadic templates and std::move() where forwarding is needed

17. Use simple metaprogramming to implement efficient and elegant unit systems (for fine-grained type checking)
18. Use user-defined literals to simplify the use of units

Chapter 29 - A Matrix Design

1. List basic use cases
2. Always provide input and output operations to simplify simple testing (e.g., unit testing)
3. Carefully list the properties a program, class, or library ideally should have
4. List the properties of a program, class, or library that are considered beyond the scope of the project
5. When designing a container template, carefully consider the requirements on the element type
6. Consider how the design might accommodate run-time checking (e.g., for debugging)
7. If possible, design a class to mimic existing professional notation and semantics
8. Make sure that the design does not leak resources (e.g., have a unique owner for each resource and use RAII)
9. Consider how a class can be constructed and copied
10. Provide complete, flexible, efficient, and semantically meaningful access to elements
11. Place implementation details in their own _impl namespace
12. Provide common operations that do not require direct access to the representation as helper functions
13. For fast access, keep data compact and use accessor objects to provide necessary nontrivial access operations
14. The structure of data can often be expressed as nested initializer lists
15. When dealing with numbers, aways consider "end cases," such as zero and "many"
16. In addition to unit testing and testing that the code meets its requirements, test the design through examples of real use
17. Consider how the design might accommodate unusually stringent performance requirements

Chapter 30 - Standard Library Summary

1. Use standard-library facilities to maintain portability
2. Use standard-library facilities to minimize maintenance costs
3. Use standard-library facilities as a base for more extensive and more specialized libraries
4. Use standard-library facilities as a model for flexible, widely usable software
5. The standard-library facilities are defined in namespace std and found in standard-library headers
6. A C standard-library header X.h is presented as a C++ standard-library header in

7. Do not try to use a standard-library facility without #include-ing its header
8. To use a range-for on a built-in array, #include
9. Prefer exception-based error handling over return-code-based error handling
10. Always catch exception& (for standard-library and language support exceptions) and ... (for unexpected exceptions)
11. The standard-library exception hierarchy can be (but does not have to be) used for a user's own exceptions
12. Call terminate() in case of serious trouble
13. Use static_assert() and assert() extensively
14. Do not assume that assert() is always evaluated
15. If you can't use exceptions, consider <system_error>

Chapter 31 - STL Containers

1. An STL container defines a sequence
2. Use vector as your default container
3. Insertion operators, such as insert() and push_back() are often more efficient on a vector than on a list
4. Use forward_list for sequences that are usually empty
5. When it comes to performance, don't trust your intuition: measure
6. Don't blindly trust asymptotic complexity measures of individual operations can vary dramatically
7. STL containers are resource handles
8. A map is usually implemented as a red-black tree
9. An unordered_map is a hash table
10. To be an element type for a STL container, a type must provide copy or move operations
11. Use containers of pointers or smart pointers when you need to preserve polymorphic behavior
12. Comparison operations should implement a strict weak order
13. Pass a container by reference and return a container by value
14. For a container, use the ()-initializer syntax for sizes and the {}-initializer syntax for lists of elements
15. For simple traversals of a container, use a range-for loop or a begin/end pair of iterators
16. Use const iterators where you don't need to modify the elements of a container
17. Use auto to avoid verbosity and typos when you use iterators
18. Use reserve() to avoid invalidating pointers and iterators to elements
19. Don't assume performance benefits from reserve() without measurement
20. Use push_back() or resize() on a container rather than realloc() on an array
21. Don't use iterators into a resized vector or deque

22. When necessary, use reserve() to make performance predictable
23. Do not assume that . range checks
24. Use at() when you need guaranteed range checks
25. Use emplace() for notational convenience
26. Prefer compact and contiguous data structures
27. Use emplace() to avoid having to pre-initialize elements
28. A list is relatively expensive to traverse
29. A list usually has a four-word-per-element memory overhead
30. The sequence of an ordered container is defined by its comparison object (by default <)
31. The sequence of an unordered container (a hashed container) is not predictably ordered
32. Use unordered containers if you need fast lookup for large amounts of data
33. Use unordered containers for element types with no natural order (e.g., no reasonable <)
34. Use ordered associative containers (e.g., map and set) if you need to iterate over their elements in order
35. Experiment to check that you have an acceptable hash function
36. Hash function obtained by combining standard hash functions for elements using exclusive or are often good
37. 0.7 is often a reasonable load factor
38. You can provide alternative interfaces for containers
39. The STL adaptors do not offer direct access to their underlying containers

Chapter 32 - STL Algorithms

1. An STL algorithm operates on one or more sequences
2. An input sequence is half-open and defined by a pair of iterators
3. When searching, an algorithm usually returns the end of the input sequence to indicate "not found"
4. Prefer a carefully specified algorithm to "random code"
5. When writing a loop, consider whether it could be expressed as a general algorithm
6. Make sure that a pair of iterator arguments really do specify a sequence
7. When the pair-of-iterators style becomes tedious, introduce a container/range algorithm
8. Use predicates and other function objects to give standard algorithms a wider range of meanings
9. A predicate must not modify its argument
10. The default == and < on pointers are rarely adequate for standard algorithms
11. Know the complexity of the algorithms you use, but remember that a complexity measure is only a rough guide to performance
12. Use for_each() and transform() only when there is no more-specific algorithm for a task
13. Algorithms do not directly add or subtract elements from their argument sequences

14. If you have to deal with uninitialized objects, consider the uninitialized_⍰⍰algorithms
15. An STL algorithm uses an equality comparison generated from its ordering comparison, rather than ==
16. Note that sorting and searching C-style strings requires the user to supply a string comparison operation

Chapter 33 - STL Iterators

1. An input sequence is defined by a pair of iterators
2. An output sequence is defined by a single iterator
3. For any iterator p, [p:p] is the empty sequence
4. Use the end of a sequence to indicate "not found"
5. Think of iterators as more general and often better behaved pointers
6. Use iterator types, such as list::iterator, rather than pointers to refer to elements of a container
7. Use iterator_traits to obtain information about iterators
8. You can do compile-time dispatch using iterator_traits
9. Use iterator_traits to select an optimal algorithm based on an iterator's category
10. iterator_traits are an implementation detail
11. Use base() to extract an iterator from a reverse_iterator
12. You can use an insert iterator to add elements to a container
13. A move_iterator can be used to make copy operations into move operations
14. Make sure that your containers can be traversed using a range-for
15. Use bind() to create variants of functions and function objects
16. Note that bind() dereferences references early
17. A mem_fn() or a lambda can be used to convert the p–>f(a) calling convention into f(p,a)
18. Use function when you need a variable that can hold a variety of callable objects

Chapter 34 - Memory and Resources

1. Use array where you need a sequence with a constexpr size
2. Prefer array over built-in arrays
3. Use bitset if you need N bits and N is not necessarily the number of bits in a built-in integer type
4. Avoid vector
5. When using pair, consider make_pair() for type deduction
6. When using tuple, consider make_tuple() for type deduction
7. Use unique_ptr to represent exclusive ownership

8. Use shared_ptr to represent shared ownership
9. Minimize the use of weak_ptrs
10. Use allocators (only) when the usual new/delete semantics is insufficient for logical or performance reasons
11. Prefer resource handles with specific semantics to smart pointers
12. Prefer unique_ptr to shared_ptr
13. Prefer smart pointers to garbage collection
14. Have a coherent and complete strategy for management of general resources
15. Garbage collection can be really useful for dealing with leaks in programs with messy pointer use
16. Garbage collection is optional
17. Don’t disguise pointers (even if you don’t use garbage collection)
18. If you use garbage collection, use declare_no_pointers() to let the garbage collector ignore data that cannot contain pointers
19. Don’t mess with uninitialized memory unless you absolutely have to

Chapter 35 - Utilities

1. Use facilities, such as steady_clock, duration, and time_point for timing
2. Prefer facilities over facilities
3. Use duration_cast to get durations in known units of time
4. Use system_clock::now() to get the current time
5. You can inquire about properties of types at compile time
6. Use move(obj) only when the value of obj cannot be used again
7. Use forward() for forwarding

Chapter 36 - Strings

1. Use character classifications rather than handcrafted checks on character ranges
2. If you implement string-like abstractions, use character_traits to implement operations on characters
3. A basic_string can be used to make strings of characters on any type
4. Use strings as variables and members rather than as base classes
5. Prefer string operations to C-style string functions
6. Return strings by value (rely on move semantics)
7. Use string::npos to indicate "the rest of the string"
8. Do not pass a nullptr to a string function expecting a C-style string
9. A string can grow and shrink, as needed

10. Use at() rather than iterators or . when you want range checking
11. Use iterators and . rather than at() when you want to optimize speed
12. If you use strings, catch length_error and out_of_range somewhere
13. Use c_str() to produce a C-style string representation of a string (only) when you have to
14. string input is type sensitive and doesn’t overflow
15. Prefer a string_stream or a generic value extraction function (such as to) over direct use of str* numeric conversion functions
16. Use the find() operations to locate values in a string (rather than writing an explicit loop)
17. Directly or indirectly, use substr() to read substrings and replace() to write substrings

Chapter 37 - Regular Expressions

1. Use regex for most conventional uses of regular expressions
2. The regular expression notation can be adjusted to match various standards
3. The default regular expression notation is that of ECMAScript
4. For portability, use the character class notation to avoid nonstandard abbreviations
5. Be restrained
6. Prefer raw string literals for expressing all but the simplest patterns
7. Note that \i allows you to express a subpattern in terms of a previous subpattern
8. Use ? to make patterns "lazy"
9. regex can use ECMAScript, POSIX, awk, grep, and egrep notation
10. Keep a copy of the pattern string in case you need to output it
11. Use regex_search() for looking at streams of characters and regex_match() to look for fixed layouts

Chapter 38 - I/O Streams

1. Define << and >> for user-defined types with values that have meaningful textual representa- tions
2. Use cout for normal output and cerr for errors
3. There are iostreams for ordinary characters and wide characters, and you can define an iostream for any kind of character
4. There are standard iostreams for standard I/O streams, files, and strings
5. Don’t try to copy a file stream
6. Binary I/O is system specific
7. Remember to check that a file stream is attached to a file before using it
8. Prefer ifstreams and ofstreams over the generic fstream
9. Use stringstreams for in-memory formatting

10. Use exceptions to catch rare bad() I/O errors
11. Use the stream state fail to handle potentially recoverable I/O errors
12. You don’t need to modify istream or ostream to add new << and >> operators
13. When implementing a iostream primitive operation, use sentry
14. Prefer formatted input over unformatted, low-level input
15. Input into strings does not overflow
16. Be careful with the termination criteria when using get(), getline(), and read()
17. By default >> skips whitespace
18. You can define a << (or a >>) so that it behaves as a virtual function based on its second operand
19. Prefer manipulators to state flags for controlling I/O
20. Use sync_with_stdio(true) if you want to mix C-style and iostream I/O
21. Use sync_with_stdio(false) to optimize iostreams
22. Tie streams used for interactive I/O
23. Use imbue() to make an iostream reflect "cultural differences" of a locale
24. width() specifications apply to the immediately following I/O operation only
25. precision() specifications apply to all following floating-point output operations
26. Floating-point format specifications (e.g., scientific) apply to all following floating-point output operations
27. #include when using standard manipulators taking arguments
28. You hardly ever need to flush()
29. Don’t use endl except possibly for aesthetic reasons
30. If iostream formatting gets too tedious, write your own manipulators
31. You can achieve the effect (and efficiency) of a ternary operator by defining a simple function object

Chapter 39 - Locales

1. Expect that every nontrivial program or system that interacts directly with people will be used in several different countries
2. Don’t assume that everyone uses the same character set as you do
3. Prefer using locales to writing ad hoc code for culture-sensitive I/O
4. Use locales to meet external (non-C++) standards
5. Think of a locale as a container of facets
6. Avoid embedding locale name strings in program text
7. Keep changes of locale to a few places in a program
8. Minimize the use of global format information
9. Prefer locale-sensitive string comparisons and sorts

10. Make facets immutable
11. Let locale handle the lifetime of facets
12. You can make your own facets
13. When writing locale-sensitive I/O functions, remember to handle exceptions from user-supplied (overriding) functions
14. Use numput if you need separators in numbers
15. Use a simple Money type to hold monetary values
16. Use simple user-defined types to hold values that require locale-sensitive I/O (rather than casting to and from values of built-in types)
17. The time_put facet can be used for both - and -style time
18. Prefer the character classification functions in which the locale is explicit

Chapter 40 - Numerics

1. Numerical problems are often subtle. If you are not 100% certain about the mathematical aspects of a numerical problem, either take expert advice, experiment, or do both
2. Use variants of numeric types that are appropriate for their use
3. Use numeric_limits to check that the numeric types are adequate for their use
4. Specialize numeric_limits for a user-defined numeric type
5. Prefer numeric_limits over limit macros
6. Use std::complex for complex arithmetic
7. Use {} initialization to protect against narrowing
8. Use valarray for numeric computation when run-time efficiency is more important than flexibility with respect to operations and element types
9. Express operations on part of an array in terms of slices rather than loops
10. Slices is a generally useful abstraction for access of compact data
11. Consider accumulate(), inner_product(), partial_sum(), and adjacent_difference() before you write a loop to compute a value from a sequence
12. Bind an engine to a distribution to get a random number generator
13. Be careful that your random numbers are sufficiently random
14. If you need genuinely random numbers (not just a pseudo-random sequence), use random_device
15. Prefer a random number class for a particular distribution over direct use of rand()

Chapter 41 - Concurrency

1. Use concurrency to improve responsiveness or to improve throughput
2. Work at the highest level of abstraction that you can afford
3. Prefer packaged_task and futures over direct use of threads and mutexes

4. Prefer mutexes and condition_variables over direct use of atomics except for simple counters
5. Avoid explicitly shared data whenever you can
6. Consider processes as an alternative to threads
7. The standard-library concurrency facilities are type safe
8. The memory model exists to save most programmers from having to think about the machine architecture level of computers
9. The memory model makes memory appear roughly as naively expected
10. Separate threads accessing separate bit-fields of a struct may interfere with each other
11. Avoid data races
12. Atomics allow for lock-free programming
13. Lock-free programming can be essential for avoiding deadlock and to ensure that every thread makes progress
14. Leave lock-free programming to experts
15. Leave relaxed memory models to experts
16. A volatile tells the compiler that the value of an object can be changed by something that is not part of the program
17. A C++ volatile is not a synchronization mechanism

Chapter 42 - Threads and Tasks

1. A thread is a type-safe interface to a system thread
2. Do not destroy a running thread
3. Use join() to wait for a thread to complete
4. Consider using a guarded_thread to provide RAII for threads
5. Do not detach() a thread unless you absolutely have to
6. Use lock_guard or unique_lock to manage mutexes
7. Use lock() to acquire multiple locks
8. Use condition_variables to manage communication among threads
9. Think in terms of tasks that can be executed concurrently, rather than directly in terms of threads
10. Value simplicity
11. Return a result using a promise and get a result from a future
12. Don’t set_value() or set_exception() to a promise twice
13. Use packaged_tasks to handle exceptions thrown by tasks and to arrange for value return
14. Use a packaged_task and a future to express a request to an external service and wait for its response
15. Don’t get() twice from a future
16. Use async() to launch simple tasks

17. Picking a good granularity of concurrent tasks is difficult: experiment and measure
18. Whenever possible, hide concurrency behind the interface of a parallel algorithm
19. A parallel algorithm may be semantically different from a sequential solution to the same problem (e.g., pfind_all() vs. find())
20. Sometimes, a sequential solution is simpler and faster than a concurrent solution

Chapter 43 - The C Standard Library

1. Use fstreams rather than fopen()/fclose() if you worry about resource leaks
2. Prefer to for reasons of type safety and extensibility
3. Never use gets() or scanf("%s",s)
4. Prefer to for reasons of ease of use and simplicity of resource management
5. Use the C memory management routines, such as memcpy(), only for raw memory
6. Prefer vector to uses of malloc() and realloc()
7. Beware that the C standard library does not know about constructors and destructors
8. Prefer to for timing
9. For flexibility, ease of use, and performance, prefer sort() over qsort()
10. Don’t use exit()
11. Don’t use longjmp()

Chapter 44 - Compatibility

1. Before using a new feature in production code, try it out by writing small programs to test the standards conformance and performance of the implementations you plan to use
2. For learning C++, use the most up-to-date and complete implementation of Standard C++ that you can get access to
3. The common subset of C and C++ is not the best initial subset of C++ to learn
4. Prefer standard facilities to nonstandard ones
5. Avoid deprecated features such as throw-specifications
6. Avoid C-style casts
7. "Implicit int" has been banned, so explicitly specify the type of every function, variable, const, etc.
8. When converting a C program to C++, first make sure that function declarations (prototypes) and standard headers are used consistently
9. When converting a C program to C++, rename variables that are C++ keywords
10. For portability and type safety, if you must use C, write in the common subset of C and C++
11. When converting a C program to C++, cast the result of malloc() to the proper type or change all uses of malloc() to uses of new

12. When converting from malloc() and free() to new and delete, consider using vector, push_back(), and reserve() instead of realloc()
13. When converting a C program to C++, remember that there are no implicit conversions from ints to enumerations
14. A facility defined in namespace std is defined in a header without a suffix (e.g., std::cout is declared in)
15. Use to get std::string (<string.h> holds the C-style string functions)
16. For each standard C header <X.h> that places names in the global namespace, the header places the names in namespace std
17. Use extern "C" when declaring C functions