SPSRB Common Standards Working Group

Standards, Guidelines and Recommendations for Writing C++ Code

Version 1.0
December, 2011

VERSION NUMBER IDENTIFIER

The document version number which also corresponds to the Document Control Number (DCN) identifies whether the document is a working copy, final, revision, or update, defined as follows:

Working Copy or Draft: a document not yet finalized or ready for distribution; sometimes called a draft. Use 0.1A, 0.1B, etc. for unpublished documents.

Final Copy: the first definitive edition of the document after it passes through the drafting stage. This first edition is always identified as Version 1.0.

Revision: an edition with minor changes from the previous edition, defined as changes affecting less than one-third of the pages in the document. The version numbers for revisions 1.1 through 1.9, 2.1 through 2.9, and so forth. After nine revisions, any other changes to the document are considered an update. A revision in draft, i.e. before being re-baselined should be numbered as 1.1A, 1.1B, etc.

Update: an edition with major changes from the previous edition, defined as changes affecting more than one-third of the pages in the document. The version number for an update is always a whole number (Version 2.0, 3.0, 4.0, and so forth).

DOCUMENT HISTORY
DOCUMENT REVISION LOG

The Document Revision Log identifies the series of revisions to this document since the baseline release. Please refer to the above page for version number information.

DOCUMENT TITLE: General Programming Principles and Guidelines			
DOCUMENT CHANGE HISTORY			
Revision No.	**Date**	**Revision Originator Project Group**	**CCR Approval # and Date**
0.1	May 2011	Initial Drafting by CSWG	
1.0	December 2011	Initial Release by CSWG	

LIST OF CHANGES

Significant alterations made to this document are annotated in the List of Changes table.

DOCUMENT TITLE: General Programming Principles and Guidelines					
LIST OF CHANGE-AFFECTED PAGES/SECTIONS/APPENDICES					
Version Number	**Date**	**Changed By**	**Page**	**Section**	**Description of Change(s)**

TABLE OF CONTENTS

1. INTRODUCTION

The National Environmental Satellite Data Distribution Service (NESDIS) develops and implements algorithms that transform environmental satellite images of the Earth into meaningful environmental data which are then employed in a full-time operational setting. In the past, software developed within NESDIS was created by the differing entities throughout the service, each creating code to fulfill various research, operational and archival needs. This meant software was written in various programming languages and idiosyncratic styles, moreover suffering from a lack of coordinating documentation in most cases. The resulting software is consequently often costly to maintain as the source code may have been mislaid, the code may be difficult to read and understand, documentation may be inadequate, or the original developers may no longer be able to maintain their code.

The purpose of developing common software programming standards is to reduce the cost of the software lifecycle and streamline the algorithm implementation process. This follows a trajectory from initial research and software development to operational use and finally through to divestiture and retirement where costs accumulate throughout the lifecycle. Implementation of these Satellite Products and Services Review Board (SPSRB) approved coding standards will shift costs away from operations and maintenance as the problems are resolved upstream. Promoting the accountability of the developers and scientists to create standardized software programs will benefit NESDIS as a whole. Higher front-end expenditure will be repaid in the form of lower operational and maintenance costs over subsequent years. It is intended that the implementation expenses of the common software standards will be funded through the Office of Systems Development (OSD) Product System Development and Implementation (PSDI) process, and must be included in relevant budgets and projects plans when applying for PSDI funds.

Having common programming standards used by all SPSRB stakeholders will aid in cross-organization communication and implementation of codes. It will also produce a software catalog that:

- Is robust
- Is readily portable (platform independent)
- Is modular and reusable
- Is inexpensive to implement and maintain operationally
- Is written in a widely used and supported language
- Has a common look and structure
- Adheres to best programming practices
- Is well documented
- Is easily readable and understandable
- Behaves in a standard manner (exception handling, file input/output)
- Uses common shared libraries

1.1 Programming Standards and Guideline Definitions

It is recognized that certain stylistic suggestions which make code easier to read (e.g. lining up attributes, or using all lower case or mixed case) are subjective and therefore should not have the same weight as techniques and practices that are known to improve code quality. For this reason, the standards within documents produced by the SPSRB Common Standards Working Group are divided into three components; Standards, Guidelines and Recommendations (SGRs):

Standard: Aimed at ensuring portability, readability and robustness. Compliance with this category is mandatory.

Guideline: Good practices. Compliance with this category is strongly encouraged. The case for deviations will need to be argued by the programmer.

Recommendation: Compliance with this category is optional, but is encouraged for consistency.

These three standards will thus be found in the above format throughout this document, indicating the weight of a particular standard. If possible, all standards, guidelines and recommendations should be followed when programming. Else, programmers should include these components whenever possible, keeping in mind their respective weight. Please refer to these definitions as needed.

1.2 Key Word Definitions

These words and phrases are vital to the comprehension of this document. These are language-specific and general terms with which a user may not be readily familiar. Listed below in alphabetical order are the terms specified by the CSWG to be helpful to users and may be referenced as necessary.

***Compilation Unit:*.** A compilation unit is the code file, which is submitted to the compiler to produce an object file. A compilation unit consists of comments, preprocessor statements (optional), declaration lists (optional), and functions. A compilation unit may also be a complete program.

Compound Statement: A compound statement consists of a beginning left brace "{", preprocessor statements (optional), a declaration list (optional), a statement list (optional), and a closing right brace "}".

Function: A function consists of a function declaration (used to name the function and declare its type), formal argument, and a compound statement.

Functional Description: A functional description is a summary of the function's purpose and consists of information needed by the user.

Line of Code: For the purposes of counting software size, a source line of code (SLOC) is defined as:

- A semicolon terminator outside of comments, parentheses and string/character literals
- Compiler directives ('#')
- One of the following statements:
  - `if`
  - `switch`
  - `while`
  - `case`
  - `for`
  - `default`

Mixed Mode: This term is used in reference to variables of differing types, which are equated, compared, or otherwise used in arithmetic expressions.

Process Description: A process description is a detailed expansion of the functional description and consists of information needed by the maintainer.

Program: A C++ program is composed of functions, one of which must be *main, and classes*. The functions may be in one or more software units.

1.3 Reference Documents

C++ Coding Standards: 101 Rules, Guidelines, and Best Practices---Herb Sutter; Andrei Alexandrescu

Cpp Coding Standards for OpenOffice.org project Found at http://wiki.services.openoffice.org/wiki/Cpp_Coding_Standards

2. Coding and Design Style

This section describes the coding and design style standards, guidelines, and recommendations of the CSWG.

2.1 Design Style

Standard: **Hide information**.

Do not expose internal information from an entity that provides an abstraction.

Guideline: **Give one entity one cohesive responsibility.**

Give each entity (variable, class, function, namespace, module, library) one well-defined responsibility. As an entity grows, its scope of responsibility naturally increases, but its responsibility should not diverge.

Recommendation: **Minimize global and shared data.**

Avoid shared data, especially global data. Shared data increases coupling which reduces maintainability and often performance.

Recommendation: **Know when and how to code for concurrency**.

If an application uses multiple threads or processes, know how to minimize sharing objects where possible and share the right ones safely.

Recommendation: **Ensure resources are owned by objects. Use explicit RAII and smart pointers**.

C++'s “resource acquisition is initialization” (RAII) idiom is the power tool for correct resource handling. RAII allows the compiler to provide strong and automated guarantees that in other languages require fragile hand-coded idioms. When allocating a raw resource, immediately pass it to an owning object. Never allocate more than one resource in a single statement.

2.2 Coding Style

Standard: **Avoid magic numbers**.

Programming isn't magic, so do not incant it: Avoid spelling literal constants like 42 or 3.14159 in code. They are not self-explanatory and complicate maintenance by adding a hard-to-detect form of duplication. Use symbolic names and expressions instead, such as width * aspectRatio.

Standard: **Avoid long functions. Avoid deep nesting**.

Markedly long functions and nested code blocks are usually caused by failing to give one function a cohesive responsibility; both are often solved by better refactoring.

Standard: **Make header files self-sufficient**.

Ensure that each header is able to compile on its own by having it include any headers its contents depend upon.

Guideline: **Use `const` proactively**.

Immutable values are easier to understand, track, and reason about, so prefer constants over variables wherever it is sensible and make const the default choice when values are defined. Generally safe, checked at compile time, and its integrated with C++'s type system, `const` is preferred. Do not cast away const except to call a const- incorrect function.

Guideline: **Avoid initialization dependencies across compilation units**.

Namespace-level objects in different compilation units should never depend on each other for initialization, because their initialization order is undefined. If not implemented, crashes and non-portability to new releases of the same compiler are prevalent.

Guideline: **Always write internal #include guards. Never write external #include guards**.

Prevent unintended multiple inclusions by using #include guards with unique names for all of header files.

```
#ifndef MyClass_hpp
#define MyClass_hpp
// Contents of MyClass.hpp header file go here…
…
}
#endif
```

Recommendation: **Prefer compile- and link-time errors to run-time errors.**

Prefer to write code that uses the compiler to check for invariants during compilation instead of checking them at runtime. Run-time checks are control- and data-dependent, which implies it is not necessarily known whether they are exhaustive. In contrast, compile-time checking is not control- or data-dependent and typically offers higher degrees of confidence.

Recommendation: **Avoid macros**.

Macros are the bluntest instrument of C and C++'s abstraction facilities and are rather difficult to use and maintain effectively. Avoid them.

Recommendation: **Minimize definitional dependencies. Avoid cyclic dependencies**.

Do not `#include` a definition when a forward declaration will do. Do not be co-dependent: cyclic dependencies occur when two modules depend directly or indirectly on one another. A module is a cohesive unit of release; modules that are interdependent are not really individual modules. Thus, cyclic dependencies work against modularity and are a bane of large projects. Avoid them.

3. Functions and Operators

This section describes the standards, guidelines, and recommendations for functions and operators.

Standard: **Avoid overloading `&&`, `||`, `or` , (comma).**

The built-in `&&` `||` , and, (comma) enjoy special treatment from the compiler. If overloaded, they become ordinary functions with very different semantics and this is a sure way to introduce subtle bugs and fragilities.

Standard: **Preserve natural semantics for overloaded operators.**

Overload operators only for good reason, and preserve natural semantics; if too difficult, a developer may be misusing operator overloading.

Guideline: **Do not write code that depends on the order of evaluation of function arguments.**

The order in which arguments of a function are evaluated is unspecified, so do not rely on a specific ordering.

Recommendation: **Prefer the canonical forms of arithmetic and assignment operators.**

When defining binary arithmetic operators, provide their assignment versions as well, and write to minimize duplication and maximize efficiency.

Recommendation: **Prefer the canonical form of ++ and --. Prefer calling the prefix forms.**

The increment and decrement operators are tricky because each has pre- and postfix forms, with slightly different semantics. Define `operator++` and `operator--` such that they mimic the behavior of their built-in counterparts. Prefer to call the prefix versions if not needed the original value.

Recommendation: **Take parameters appropriately by value, (smart) pointer, or reference.**

Distinguish among input, output, and input/output parameters, and between value and reference parameters. Take them appropriately.

Recommendation: **Consider overloading to avoid implicit type conversions.**

Do not multiply objects beyond necessity. Implicit type conversions provide yntactic convenience. But when the work of creating temporary objects is unnecessary and optimization is appropriate, one can provide overloaded functions with signatures that match common argument types exactly and will not cause conversions.

4. Class Design and Inheritance

This section describes the standards, guidelines, and recommendations for class design and inheritance.

4.1 Class Design

Standard: **Do not give away specific internals.**

Avoid returning handles to internal data managed by the developer's class, so clients will not uncontrollably modify state that an object thinks it owns**.**

Standard: **Always provide new and delete together.**

Every class-specific overload `void*` operator `new(parms)` must be accompanied by a corresponding overload void operator `delete(void*, parms)` , where `parms` is a list of extra parameter types (of which the first is always `std: : size_t`). The same goes for the array forms `new()` and `delete()`.

Guideline: **Do not use dynamic memory without a reason.**

Using local object variables is faster, more concise, and less bug-prone than using "new" to create an object that is "deleted" later in the same scope.

```
{
    MyObject foo(initializer);
    // No new, no delete, no chance to leak memory if an
    // exception occurs
    foo.whatever();
    …
}
```

Guideline: **Avoid providing implicit conversions.**

Implicit conversions can often do more damage than good. Only when needed, provide implicit conversions to and from the types defined by the user, and prefer to rely on explicit conversions (explicit constructors and named conversion functions).

Guideline: **Make data members private, except in behavior-less aggregates (C- style structs).**

Keep data members private. Only in the case of simple C-style struct types that aggregate a bunch of values but do not pretend to encapsulate or provide behavior, make all data public. Avoid mixes of public and nonpublic data, which almost always signal a muddled design.

Recommendation: **Prefer minimal classes to monolithic classes.**

Small classes are easier to write, get right, test, and use. They are also more likely to be usable in a variety of situations. Prefer such small classes that embody simple concepts instead of comprehensive classes that implement many complex concepts.

Recommendation: **Practice safe overriding.**

When overriding a virtual function, preserve substitutability; in particular, observe the function's pre- and post-conditions in the base class. Do not change default arguments of virtual functions. Prefer explicitly re-declaring overrides as virtual. Beware of hiding overloads in the base class.

Recommendation: **Consider making virtual functions nonpublic, and public functions non-virtual.**

Prefer to make public functions non-virtual. Prefer to make virtual functions private, or protected if derived classes need to be able to call the base versions. (Note that this advice does not apply to destructors.)

Recommendation: **Prefer writing nonmember non-friend functions.**

Where possible, prefer making functions nonmember non-friends.

Recommendation: **If any class-specific new is provided then give all of the standard forms (plain, in-place, and nothrow).**

If a class defines any overload of operator new, it should provide overloads of all three of plain, in-place, and non-throwing operator new. Else, they will be hidden and unavailable to users of this class.

4.2 Inheritance

Guideline: **Avoid inheriting from classes that were not designed to be base classes.**

Classes meant to be used standalone obey a different blueprint than base classes. Using standalone class as a base is a serious design error and should be avoided. To add behavior, prefer to add nonmember functions instead of member functions. To add state, prefer composition instead of inheritance. Avoid inheriting from concrete base classes.

Guideline: **The virtual keyword shall be used to identify a modified inherited method.**

Recommendation: **Public inheritance is substitutability; inherit to be reused not to explicitly reuse.**

Public inheritance allows a pointer or reference to the base class to actually refer to an object of some derived class, without destroying code correctness and without needing to change existing code. Do not inherit publicly to reuse code (that exists in the base class); inherit publicly in order to be reused (by existing code that already uses base objects polymorphically).

Recommendation: **Prefer composition to inheritance.**

Inheritance is the second-tightest coupling relationship in C++, second only to friendship. Tight coupling should be avoided where possible. Therefore, prefer composition to inheritance unless the latter truly benefits the design**.**

Recommendation: **Prefer providing abstract interfaces.**

Prefer to design hierarchies that implement abstract interfaces that model abstract concepts.

5. Construction, Destruction, and Copying

This section contains standards, guidelines, and recommendations for implementing constructors, copy constructors, and destructors.

Standard: **Explicitly Define Constructor, Destructor, Copying Constructor and Assignment Operator**

The compiler will implicitly generate default versions if these are not explicitly defined. The best practice is to disallow the compiler to generate default code.

Standard: **Destructors, deallocation, and swap never fail.**

Never allow an error to be reported from a destructor, a resource deallocation function (e.g., operator delete), or a swap function. Specifically, types whose destructors may throw an exception are flatly forbidden from use with the C++ standard library.

Standard: **Copy and destroy consistently.**

If a developer defines any of the copy constructor, copy assignment operator, or destructor, then the developer needs to define one or both of the others.

Guideline: **Explicitly enable or disable copying.**

Knowingly choose among using the compiler-generated copy constructor and assignment operator, writing individual versions, or explicitly disabling both if copying should not be allowed.

```
// If the choice is to disable:
class MyClass {
   private:
     // disable copy constructor and assignment by declaring
     // them private and leaving their implementation undefined
     MyClass operator=(const MyClass& other);
     MyClass(const MyClass& other);
}
```

Guideline: **Avoid slicing. Consider Clone instead of copying in base classes.**

Object slicing is automatic, invisible, and likely to bring wonderful polymorphic designs to a screeching halt. In base classes, consider disabling the copy constructor and copy assignment operator, and instead supplying a virtual Clone member function if clients need to make polymorphic (complete, deep) copies.

Guideline: **Avoid calling virtual functions in constructors and destructors.**

Inside constructors and destructors, they do not. Worse, any direct or indirect call to an unimplemented pure virtual function from a constructor or destructor results in undefined behavior. If a design of virtual dispatch into a derived class from a base class constructor or destructor is desired, a developer needs other techniques such as post-constructors.

Guideline: **Make base class destructors public and virtual, or protected and nonvirtual.**
If deletion through a pointer to a base, a base should be allowed, thus its destructor must be public and virtual. Otherwise, it should be protected and non-virtual.

Guideline: **Define and initialize member variables in the same order.**
Member variables are always initialized in the order they are declared in the class definition; the order in which they are written in the constructor initialization list is ignored. Make sure the constructor code does not specify a different order.

Recommendation: **Prefer initialization to assignment in constructors.**
In constructors, using initialization instead of assignment to set member variables prevents needless run-time work and takes the same amount of typing.

```
class MyClass {
    MyClass(int i) :
        m1(i),
        m2(j*2)
    {
       // instead of m1 = j; m2 = j*2;
    }
    …
}
```

Recommendation: **Prefer the canonical form of assignment.**
When implementing `operator=`, prefer the canonical form—non-virtual and with a specific signature.

Recommendation: **Whenever it makes sense, provide a no-fail swap, providing it correctly.**
Consider providing a swap function to efficiently and infallibly swap the internals of this object the internals of another. Such a function can be handy for implementing a number of idioms, from smoothly moving objects around to implementing assignment easily to providing a guaranteed commit function that enables strongly error-safe calling code.

Recommendation: **When possible, constructors should initialize each data member in the class to a valid default value.**
The object should be completely initialized.

Recommendation: **Members should appear in an initialization list in the order in which they are declared, one per line and equally indented**.

6. Namespaces and Modules

This section describes the standards, guidelines, and recommendations when creating namespaces, header files, and modules.

Standard: **Do not define entities with linkage in a header file.**

Entities with linkage, including namespace-level variables or functions, have memory allocated for them. Defining such entities in header files creates either link-time errors or memory waste; put all entities with linkage in implementation files.

Standard: **Do not allow exceptions to propagate across module boundaries.**

There is no ubiquitous binary standard for C++ exception handling. Do not allow exceptions to propagate between two pieces of code unless the compiler is explicitly controlled by the developer and compiler options used to build both sides. Else, the modules might not support compatible implementations for exception propagation. In other words, do not let exceptions propagate across module/subsystem boundaries.

Guideline: **Keep a type and its nonmember function interface in the same namespace.**

Nonmember functions that are designed to be part of the interface of a class X (notably operators and helper functions) must be defined in the same namespace as the X in order to be called correctly.

Guideline: **Keep types and functions in separate namespaces unless they are specifically intended to work together.**

Isolate types from unintentional argument-dependent lookup (ADL, also known as Koenig lookup), and encourage intentional ADL, by putting them in their own namespaces (along with their directly related nonmember functions. Avoid putting a type into the same namespace as a function from a template or an operator.

Guideline: **Do not write namespaces in a header file or before an `#include`.**

Never write a using declaration or a using directive before an `#include` directive. Corollary: In header files, do not write namespace-level using directives or using declarations; instead, explicitly namespace-qualify all names. (The second rule follows from the first, because headers can never know what other header `#includes` might appear after them.)

Guideline: **Avoid allocating and deallocating memory in different modules.**

Allocating memory in one module and deallocating it in a different module makes a program fragile by creating a subtle long-distance dependency between those modules. They must be compiled with the same compiler version and same flags (notably debug vs. `NDEBUG`) and the same standard library implementation, and in practice the module allocating the memory had better still be loaded when the deallocation happens.

Guideline: **Use sufficiently portable types in a module's interface.**

Do not allow a type to appear in a module's external interface unless there is a way to ensure that all clients understand the type correctly. Use the highest level of abstraction that clients can understand.

7. Templates and Genericity

This section describes the guidelines and recommendations for creating templates and generic programming.

Guideline: **Do not specialize function templates.**

When extending someone else's function template (including `std: : swap`), avoid trying to write a specialization; instead, write an overload of the function template, and put it in the namespace of the type(s) the overload is designed to be used for. Avoid encouraging direct specialization of the function template itself.

Guideline: **Do not write unintentionally non-generic code.**

Use the most generic and abstract means to implement a piece of functionality.

Recommendation: **Blend static and dynamic polymorphism judiciously.**

Static and dynamic polymorphism are complementary; understand the tradeoffs, use each for what its best at, and mix them to get the best of both worlds.

Recommendation **Customize intentionally and explicitly.**

When writing a template, provide points of customization knowingly and correctly, and document them clearly. When using a template, know how the template intends for users to customize it for each particular use and customize it appropriately.

8. Error Handling and Exceptions

This section describes the standards, guidelines, and recommendations to employ when using and creating error handling and exceptions.

Standard: **Avoid exception specifications.**

Do not write exception specifications on functions unless forced to, because other code cannot change what has already introduced them.

Guideline: **Assert liberally to document internal assumptions and invariants.**

Use assert or an equivalent liberally to document assumptions internal to a module (i.e., where the caller and callee are maintained by the same person or team) that must always be true and otherwise represent programming errors (e.g., violations of the post-conditions of a function are detected by the caller of the function). Ensure that assertions do not perform side effects.

Guideline: **Throw by value, catch by reference.**

Throw exceptions by value (not pointer) and catch them by reference (usually to const). This is the combination that meshes best with exception semantics. When rethrowing the same exception, prefer just `throw;` to `throw e;`.

Recommendation: **Establish a rational error handling policy, and follow it strictly.**

Develop a practical, consistent, and rational error handling policy early in design, and then continue this policy. Ensure that it includes:

- In what conditions are errors present.
- How important or urgent each error is.
- Which code is responsible for detecting the error.
- What mechanisms are used to report and propagate error notifications in each module.
- What code is responsible for doing something about the error.
- How the error will be logged or users notified.
- Change error handling mechanisms only on module boundaries.

Recommendation: **Distinguish between errors and non-errors.**

A function is a unit of work. Thus, failures should be viewed as errors or otherwise based on their impact on functions. Within a function `f`, a failure is an error if and only if it violates one of `f`'s preconditions or prevents f from meeting any of its callees' preconditions, achieving any of `f`'s own postconditions, or reestablishing any invariant that `f` shares responsibility for maintaining.

Recommendation: **Prefer to use exceptions to report errors.**

Prefer using exceptions over error codes to report errors. Use status codes (e.g., return codes, errno) for errors when exceptions cannot be used and for conditions that are not errors. Use other methods, such as graceful or ungraceful termination, when recovery is impossible or not required**.**

Recommendation: **Report, handle, and translate errors appropriately.**

Report errors at the point they are detected and identified as errors. Handle or translate each error at the nearest level that can do it correctly.

9. Containers and Algorithms

This section contains guidelines and recommendations for creating and using containers and algorithms.

9.1 Containers

Guideline: **Use `vector` by default. Otherwise, choose an appropriate container.**

If there is a good reason to use a specific container type, use that container type. Otherwise, write vector. They can both be the 'right thing to do.'

Guideline: **Use `vector` and `string` instead of arrays.**

Avoid implementing array abstractions with C-style arrays, pointer arithmetic, and memory management primitives. Using vector or string not only makes coding easier, but also aids developers in writing safer and more scalable software.

Guideline: **Use `vector` (and `string::c_str`) to exchange data with non-C++ Application programming interfaces (APIs).**

`Vector` and `string: : c_str` are gateways to communicate with non-C++ APIs. But do not assume iterators are pointers; get the address of the element referred to by a `vector<T>: : iterator iter`, `use &*iter.`

Guideline: **Store only values and smart pointers in containers.**

Containers assume they contain value-like types, including value types (held directly), smart pointers, and iterators.

Recommendation: **Prefer `push_back` to other ways of expanding a sequence.**

If one does not need to care about the insert position, prefer using `push_back` to add an element to sequence. Other means can be both vastly slower and less clear.

Recommendation: **Prefer range operations to single-element operations.**

When adding elements to sequence containers, prefer a range operations (e.g., the form of insert that takes a pair of iterators) instead of a series of calls to the single-element form of the operation. Calling the range operation is generally easier to write, easier to read, and more efficient than an explicit loop.

9.2 Algorithms

Guideline: **Use a checked Standard Template Library (STL) implementation.**

Use a checked STL implementation, even if it is only available for one compiler platforms, and even if it is only used during pre-release testing.

Recommendation: **Prefer algorithm calls to handwritten loops.**

For very simple loops, handwritten loops can be the simplest and most efficient solution. But writing algorithm calls instead of handwritten loops can be more expressive and maintainable, less error-prone, and as efficient. When calling algorithms, consider writing custom function objects that encapsulate the logic needed. Avoid cobbling together parameter-binders and simple function objects (e.g., `bind2nd plus`), which usually degrade clarity. Consider trying the [Boost]

Lambda library, which automates the task of writing function objects.

Recommendation: **Use the right STL search algorithm.**

This item applies to searching for a particular value in a range, or for the location where it would be if it were in the range. To search an unsorted range, use `find/find_`if or `count/count_if`. To search a sorted range, use `lower_bound upper_bound equal_range`, or (rarely) `binary_search`. Despite its common name, `binary_search` is usually not the right choice.

Recommendation: **Use the right STL sort algorithm.**

Understand what each of the sorting algorithms does, and use the cheapest algorithm that does what is needed.

Recommendation: **Prefer function objects over functions as algorithm and comparer arguments.**

Prefer passing function objects, not functions, to algorithms. Comparers for associative containers must be function objects. Function objects are adaptable and, counterintuitively, they typically produce faster code than functions.

10. Type Safety

This section describes the guidelines and recommendations for ensuring type safety.

Guideline: **Avoid type switching; prefer polymorphism.**

Avoid switching on the type of an object to customize behavior. Use templates and virtual functions to let types (not their calling code) decide their behavior.

Guideline: **Avoid using `reinterpret_cast`.**

Do not try to use `reinterpret_cast` to force the compiler to reinterpret the bits of an object of one type as being the bits of an object of a different type. That's the opposite of maintaining type safety, and `reinterpret_cast` is not even guaranteed to do that or anything else in particular. Consider using `htonl()` or the like if reading or writing binary numeric data, or just process I/O byte by byte. The code is already broken if it would break on a machine with opposite byte order.

Guideline: **Avoid using `static_cast` on pointers.**

Safe alternatives range from using `dynamic_cast` to refactoring to redesigning.

Guideline **Avoid casting away `const`.**

Casting away `const` sometimes results in undefined behavior, and indicates poor programming style even when legal.

Guideline: **Do not use C-style casts.**

C-style casts have different (and often dangerous) semantics depending on context, all disguised behind a single syntax. Replacing C-style casts with C++-style casts helps guard against unexpected errors.

Guideline: **Do not use `memcpy` or `memcmp`.**

Do not use `memcpy` and `memcmp` to copy or compare anything more structured than raw memory.

Guideline: **Do not use unions to reinterpret representation.**

Unions can be abused into obtaining a "cast without a cast" by writing one member and reading another. This is even less predictable than `reinterpret_cast`.

Guideline: **Do not use `varargs` (ellipsis).**

The ellipsis is a dangerous carryover from C. Avoid `varargs`, and use higher-level C++ constructs and libraries instead.

Guideline: **Do not treat arrays polymorphically.**

Treating arrays polymorphically is a gross type error that a compiler will probably remain silent about.

Recommendation: **Rely on types, not on representations.**

Do not make assumptions about how objects are exactly represented in memory. Instead, let types decide how their objects are written to and read from memory.