C++ Core Guidelines

February 15, 2016

Editors:

- Bjarne Stroustrup
- Herb Sutter

This document is a very early draft. It is inkorrekt, incompleat, and pÂµÃ¸oorly formatted. Had it been an open source (code) project, this would have been release 0.6. Copying, use, modification, and creation of derivative works from this project is licensed under an MIT-style license. Contributing to this project requires agreeing to a Contributor License. See the accompanying LICENSE file for details. We make this project available to “friendly users” to use, copy, modify, and derive from, hoping for constructive input.

Comments and suggestions for improvements are most welcome. We plan to modify and extend this document as our understanding improves and the language and the set of available libraries improve. When commenting, please note the introduction that outlines our aims and general approach. The list of contributors is here.

Problems:

- The sets of rules have not been thoroughly checked for completeness, consistency, or enforceability.
- Triple question marks (???) mark known missing information
- Update reference sections; many pre-C++11 sources are too old.
- For a more-or-less up-to-date to-do list see: To-do: Unclassified proto-rules

You can read an explanation of the scope and structure of this Guide or just jump straight in:

- SL: The Standard library
- SF: Source files
- CPL: C-style programming
- PRO: Profiles
- GSL: Guideline support library
- FAQ: Answers to frequently asked questions

Supporting sections:

- NL: Naming and layout
- PER: Performance
- N: Non-Rules and myths
- RF: References
- Appendix A: Libraries
- Appendix B: Modernizing code
- Appendix C: Discussion
- Glossary
- To-do: Unclassified proto-rules

or look at a specific language feature

- assignment
- `class`
- constructor
- derived `class`
- destructor
- exception
- `for`
- `inline`
- initialization
- lambda expression
- operator
- `public`, `private`, and `protected`
- `static_assert`
- `struct`
- `template`
- `unsigned`
- `virtual`

Definitions of terms used to express and discuss the rules, that are not language-technical, but refer to design and programming techniques

- error
- exception
- failure
- invariant
- leak
- precondition
- postcondition
- resource
- exception guarantee

Abstract

This document is a set of guidelines for using C++ well. The aim of this document is to help people to use modern C++ effectively. By "modern C++" we mean C++11 and C++14 (and soon C++17). In other words, what would you like your code to look like in 5 years' time, given that you can start now? In 10 years' time?

The guidelines are focused on relatively higher-level issues, such as interfaces, resource management, memory management, and concurrency. Such rules affect application architecture and library design. Following the rules will lead to code that is statically type safe, has no resource leaks, and catches many more programming logic errors than is common in code today. And it will run fast - you can afford to do things right.

We are less concerned with low-level issues, such as naming conventions and indentation style. However, no topic that can help a programmer is out of bounds.

Our initial set of rules emphasizes safety (of various forms) and simplicity. They may very well be too strict. We expect to have to introduce more exceptions to better accommodate real-world needs. We also need more rules.

You will find some of the rules contrary to your expectations or even contrary to your experience. If we haven't suggested you change your coding style in any way, we have failed! Please try to verify or disprove rules! In particular, we'd really like to have some of our rules backed up with measurements or better examples.

You will find some of the rules obvious or even trivial. Please remember that one purpose of a guideline is to help someone who is less experienced or coming from a different background or language to get up to speed.

Many of the rules are designed to be supported by an analysis tool. Violations of rules will be flagged with references (or links) to the relevant rule. We do not expect you to memorize all the rules before trying to write code. One way of thinking about these guidelines is as a specification for tools that happens to be readable by humans.

The rules are meant for gradual introduction into a code base. We plan to build tools for that and hope others will too.

Comments and suggestions for improvements are most welcome. We plan to modify and extend this document as our understanding improves and the language and the set of available libraries improve.

In: Introduction

This is a set of core guidelines for modern C++, C++14, and taking likely future enhancements and taking ISO Technical Specifications (TSs) into account. The aim is to help C++ programmers to write simpler, more efficient, more maintainable code.

Introduction summary:

- In.target: Target readership
- In.aims: Aims
- In.not: Non-aims
- In.force: Enforcement
- In.struct: The structure of this document
- In.sec: Major sections

In.target: Target readership

All C++ programmers. This includes programmers who might consider C.

In.aims: Aims

The purpose of this document is to help developers to adopt modern C++ (C++11, C++14, and soon C++17) and to achieve a more uniform style across code bases.

We do not suffer the delusion that every one of these rules can be effectively applied to every code base. Upgrading old systems is hard. However, we do believe that a program that uses a rule is less error-prone and more maintainable than one that does not. Often, rules also lead to faster/easier initial development. As far as we can tell, these rules lead to code that performs as well or better than older, more conventional techniques; they are meant to follow the zero-overhead principle ("what you don't use, you don't pay for" or "when you use an abstraction mechanism appropriately, you get at least as good performance as if you had handcoded using lower-level language constructs"). Consider these rules ideals for new code, opportunities to exploit when working on older code, and try to approximate these ideas as closely as feasible. Remember:

In.0: Don't panic!

Take the time to understand the implications of a guideline rule on your program.

These guidelines are designed according to the "subset of a superset" principle (Stroustrup05). They do not simply define a subset of C++ to be used (for reliability, safety, performance, or whatever). Instead, they strongly recommend the use of a few simple "extensions" (library components) that make the use of the most error-prone features of C++ redundant, so that they can be banned (in our set of rules).

The rules emphasize static type safety and resource safety. For that reason, they emphasize possibilities for range checking, for avoiding dereferencing `nullptr`, for avoiding dangling pointers, and the systematic use of exceptions (via RAII). Partly to achieve that and partly to minimize obscure code as a source of errors, the rules also emphasize simplicity and the hiding of necessary complexity behind well-specified interfaces.

Many of the rules are prescriptive. We are uncomfortable with rules that simply state "don't do that!" without offering an alternative. One consequence of that is that some rules can be supported only by heuristics, rather than precise and mechanically verifiable checks. Other rules articulate general principles. For these more general rules, more detailed and specific rules provide partial checking.

These guidelines address the core of C++ and its use. We expect that most large organizations, specific application areas, and even large projects will need further rules, possibly further restrictions, and further library support. For example, hard real-time programmers typically can't use free store (dynamic memory) freely and will be restricted in their choice of libraries. We encourage the development of such more specific rules as addenda to these core guidelines. Build your ideal small foundation library and use that, rather than lowering your level of programming to glorified assembly code.

The rules are designed to allow gradual adoption.

Some rules aim to increase various forms of safety while others aim to reduce the likelihood of accidents, many do both. The guidelines aimed at preventing accidents often ban perfectly legal C++. However, when there are two ways of expressing an idea and one has shown itself a common source of errors and the other has not, we try to guide programmers towards the latter.

In.not: Non-aims

The rules are not intended to be minimal or orthogonal. In particular, general rules can be simple, but unenforceable. Also, it is often hard to understand the implications of a general rule. More specialized rules are often easier to understand and to enforce, but without general rules, they would just be a long list of special cases. We provide rules aimed at helping novices as well as rules supporting expert use. Some rules can be completely enforced, but others are based on heuristics.

These rules are not meant to be read serially, like a book. You can browse through them using the links. However, their main intended use is to be targets for tools. That is, a tool looks for violations and the tool returns links to violated rules. The rules then provide reasons, examples of potential consequences of the violation, and suggested remedies.

These guidelines are not intended to be a substitute for a tutorial treatment of C++. If you need a tutorial for some given level of experience, see the references.

This is not a guide on how to convert old C++ code to more modern code. It is meant to articulate ideas for new code in a concrete fashion. However, see the modernization section for some possible approaches to modernizing/rejuvenating/upgrading. Importantly, the rules support gradual adoption: It is typically infeasible to convert all of a large code base at once.

These guidelines are not meant to be complete or exact in every language-technical detail. For the final word on language definition issues, including every exception to general rules and every feature, see the ISO C++ standard.

The rules are not intended to force you to write in an impoverished subset of C++. They are *emphatically* not meant to define a, say, Java-like subset of C++. They are not meant to define a single "one true C++" language. We value expressiveness and uncompromised performance.

The rules are not value-neutral. They are meant to make code simpler and more correct/safer than most existing C++ code, without loss of performance. They are meant to inhibit perfectly valid C++ code that correlates with errors, spurious complexity, and poor performance.

In.force: Enforcement

Rules with no enforcement are unmanageable for large code bases. Enforcement of all rules is possible only for a small weak set of rules or for a specific user community. But we want lots of rules, and we want rules that everybody can use. But different people have different needs. But people don't like to read lots of rules. But people can't remember many rules. So, we need subsetting to meet a variety of needs. But arbitrary subsetting leads to chaos: We want guidelines that help a lot of people, make code more uniform, and strongly encourage people to modernize their code. We want to encourage best practices, rather than leave all to individual choices and management pressures. The ideal is to use all rules; that gives the greatest benefits.

This adds up to quite a few dilemmas. We try to resolve those using tools. Each rule has an **Enforcement** section listing ideas for enforcement. Enforcement might be by code review, by static analysis, by compiler, or by run-time checks. Wherever possible, we prefer "mechanical" checking (humans are slow, inaccurate, and bore easily) and static checking. Run-time checks are suggested only rarely where no alternative exists; we do not want to introduce "distributed fat". Where appropriate, we label a rule (in the **Enforcement** sections) with the name of groups of related rules (called "profiles"). A rule can be part of several profiles, or none. For a start, we have a few profiles corresponding to common needs (desires, ideals):

- **type**: No type violations (reinterpreting a `T` as a `U` through casts/unions/varargs)
- **bounds**: No bounds violations (accessing beyond the range of an array)

- **lifetime**: No leaks (failing to `delete` or multiple `delete`) and no access to invalid objects (dereferencing `nullptr`, using a dangling reference).

The profiles are intended to be used by tools, but also serve as an aid to the human reader. We do not limit our comment in the **Enforcement** sections to things we know how to enforce; some comments are mere wishes that might inspire some tool builder.

Tools that implement these rules shall respect the following syntax to explicitly suppress a rule:

```
[[suppress(tag)]]
```

where "tag" is the anchor name of the item where the Enforcement rule appears (e.g., for C.134 it is "Rh-public"), the name of a profile group-of-rules ("type", "bounds", or "lifetime"), or a specific rule in a profile ("type.4", or "bounds.2").

In.struct: The structure of this document

Each rule (guideline, suggestion) can have several parts:

- The rule itself - e.g., **no naked `new`**
- A rule reference number - e.g., **C.7** (the 7th rule related to classes). Since the major sections are not inherently ordered, we use a letter as the first part of a rule reference "number". We leave gaps in the numbering to minimize "disruption" when we add or remove rules.
- **Reason**s (rationales) - because programmers find it hard to follow rules they don't understand
- **Example**s - because rules are hard to understand in the abstract; can be positive or negative
- **Alternative**s - for "don't do this" rules
- **Exception**s - we prefer simple general rules. However, many rules apply widely, but not universally, so exceptions must be listed
- **Enforcement** - ideas about how the rule might be checked "mechanically"
- **See also**s - references to related rules and/or further discussion (in this document or elsewhere)
- **Note**s (comments) - something that needs saying that doesn't fit the other classifications
- **Discussion** - references to more extensive rationale and/or examples placed outside the main lists of rules

Some rules are hard to check mechanically, but they all meet the minimal criteria that an expert programmer can spot many violations without too much trouble. We hope that "mechanical" tools will improve with time to approximate what such an expert programmer notices. Also, we assume that the rules will be refined over time to make them more precise and checkable.

A rule is aimed at being simple, rather than carefully phrased to mention every alternative and special case. Such information is found in the **Alternative** paragraphs and the Discussion sections. If you don't understand a rule or disagree with it, please visit its **Discussion**. If you feel that a discussion is missing or incomplete, enter an Issue explaining your concerns and possibly a corresponding PR.

This is not a language manual. It is meant to be helpful, rather than complete, fully accurate on technical details, or a guide to existing code. Recommended information sources can be found in the references.

In.sec: Major sections

- In: Introduction
- P: Philosophy
- I: Interfaces
- F: Functions
- C: Classes and class hierarchies
- Enum: Enumerations
- R: Resource management
- ES: Expressions and statements
- E: Error handling
- Con: Constants and immutability
- T: Templates and generic programming
- CP: Concurrency
- SL: The Standard library
- SF: Source files
- CPL: C-style programming
- PRO: Profiles
- GSL: Guideline support library
- FAQ: Answers to frequently asked questions

Supporting sections:

- NL: Naming and layout
- PER: Performance
- N: Non-Rules and myths
- RF: References
- Appendix A: Libraries
- Appendix B: Modernizing code
- Appendix C: Discussion

- Glossary
- To-do: Unclassified proto-rules

These sections are not orthogonal.

Each section (e.g., "P" for "Philosophy") and each subsection (e.g., "C.hier" for "Class Hierarchies (OOP)") have an abbreviation for ease of searching and reference. The main section abbreviations are also used in rule numbers (e.g., "C.11" for "Make concrete types regular").

P: Philosophy

The rules in this section are very general.

Philosophy rules summary:

- P.1: Express ideas directly in code
- P.2: Write in ISO Standard C++
- P.3: Express intent
- P.4: Ideally, a program should be statically type safe
- P.5: Prefer compile-time checking to run-time checking
- P.6: What cannot be checked at compile time should be checkable at run time
- P.7: Catch run-time errors early
- P.8: Don't leak any resources
- P.9: Don't waste time or space
- P.10: Prefer immutable data to mutable data

Philosophical rules are generally not mechanically checkable. However, individual rules reflecting these philosophical themes are. Without a philosophical basis the more concrete/specific/checkable rules lack rationale.

P.1: Express ideas directly in code

Reason Compilers don't read comments (or design documents) and neither do many programmers (consistently). What is expressed in code has defined semantics and can (in principle) be checked by compilers and other tools.

Example

```
class Date {
    // ...
public:
```

```
    Month month() const;  // do
    int month();          // don't
    // ...
};
```

The first declaration of `month` is explicit about returning a `Month` and about not modifying the state of the `Date` object. The second version leaves the reader guessing and opens more possibilities for uncaught bugs.

Example

```
void do_something(vector<string>& v)
{
    string val;
    cin >> val;
    // ...
    int index = -1;             // bad
    for (int i = 0; i < v.size(); ++i)
        if (v[i] == val) {
            index = i;
            break;
        }
    // ...
}
```

That loop is a restricted form of `std::find`. A much clearer expression of intent would be:

```
void do_something(vector<string>& v)
{
    string val;
    cin >> val;
    // ...
    auto p = find(begin(v), end(v), val);  // better
    // ...
}
```

A well-designed library expresses intent (what is to be done, rather than just how something is being done) far better than direct use of language features.

A C++ programmer should know the basics of the standard library, and use it where appropriate. Any programmer should know the basics of the foundation libraries of the project being worked on, and use them appropriately. Any programmer using these guidelines should know the guideline support library, and use it appropriately.

Example

```
change_speed(double s);   // bad: what does s signify?
// ...
change_speed(2.3);
```

A better approach is to be explicit about the meaning of the double (new speed or delta on old speed?) and the unit used:

```
change_speed(Speed s);    // better: the meaning of s is specified
// ...
change_speed(2.3);        // error: no unit
change_speed(23m / 10s);  // meters per second
```

We could have accepted a plain (unit-less) `double` as a delta, but that would have been error-prone. If we wanted both absolute speed and deltas, we would have defined a `Delta` type.

Enforcement Very hard in general.

- use `const` consistently (check if member functions modify their object; check if functions modify arguments passed by pointer or reference)
- flag uses of casts (casts neuter the type system)
- detect code that mimics the standard library (hard)

P.2: Write in ISO Standard C++

Reason This is a set of guidelines for writing ISO Standard C++.

Note There are environments where extensions are necessary, e.g., to access system resources. In such cases, localize the use of necessary extensions and control their use with non-core Coding Guidelines. If possible, build interfaces that encapsulate the extensions so they can be turned off or compiled away on systems that do not support those extensions.

Extensions often do not have rigorously defined semantics. Even extensions that are common and implemented by multiple compilers may have slightly different behaviors and edge case behavior as a direct result of *not* having a rigorous standard definition. With sufficient use of any such extension, expected portability will be impacted.

Note There are environments where restrictions on use of standard C++ language or library features are necessary, e.g., to avoid dynamic memory allocation as required by aircraft control software standards. In such cases, control their (dis)use with an extension of these Coding Guidelines customized to the specific environment.

Enforcement Use an up-to-date C++ compiler (currently C++11 or C++14) with a set of options that do not accept extensions.

P.3: Express intent

Reason Unless the intent of some code is stated (e.g., in names or comments), it is impossible to tell whether the code does what it is supposed to do.

Example

```
int i = 0;
while (i < v.size()) {
    // ... do something with v[i] ...
}
```

The intent of "just" looping over the elements of `v` is not expressed here. The implementation detail of an index is exposed (so that it might be misused), and `i` outlives the scope of the loop, which may or may not be intended. The reader cannot know from just this section of code.

Better:

```
for (const auto& x : v) { /* do something with x */ }
```

Now, there is no explicit mention of the iteration mechanism, and the loop operates on a reference to `const` elements so that accidental modification cannot happen. If modification is desired, say so:

```
for (auto& x : v) { /* do something with x */ }
```

Sometimes better still, use a named algorithm:

```
for_each(v, [](int x) { /* do something with x */ });
for_each(parallel.v, [](int x) { /* do something with x */ });
```

The last variant makes it clear that we are not interested in the order in which the elements of `v` are handled.

A programmer should be familiar with

- The guideline support library
- The ISO C++ standard library
- Whatever foundation libraries are used for the current project(s)

Note Alternative formulation: Say what should be done, rather than just how it should be done.

Note Some language constructs express intent better than others.

Example If two `int`s are meant to be the coordinates of a 2D point, say so:

```
drawline(int, int, int, int);  // obscure
drawline(Point, Point);        // clearer
```

Enforcement Look for common patterns for which there are better alternatives

- simple `for` loops vs. range-`for` loops
- `f(T*, int)` interfaces vs. `f(span<T>)` interfaces
- loop variables in too large a scope
- naked `new` and `delete`
- functions with many arguments of built-in types

There is a huge scope for cleverness and semi-automated program transformation.

P.4: Ideally, a program should be statically type safe

Reason Ideally, a program would be completely statically (compile-time) type safe. Unfortunately, that is not possible. Problem areas:

- unions
- casts
- array decay
- range errors
- narrowing conversions

Note These areas are sources of serious problems (e.g., crashes and security violations). We try to provide alternative techniques.

Enforcement We can ban, restrain, or detect the individual problem categories separately, as required and feasible for individual programs. Always suggest an alternative. For example:

- unions - use `variant`
- casts - minimize their use; templates can help
- array decay - use `span`
- range errors - use `span`
- narrowing conversions - minimize their use and use `narrow` or `narrow_cast` where they are necessary

P.5: Prefer compile-time checking to run-time checking

Reason Code clarity and performance. You don't need to write error handlers for errors caught at compile time.

Example

```
void initializer(Int x)
// Int is an alias used for integers
{
    static_assert(sizeof(Int) >= 4);    // do: compile-time check

    int bits = 0;         // don't: avoidable code
    for (Int i = 1; i; i <<= 1)
        ++bits;
    if (bits < 32)
        cerr << "Int too small\n";

    // ...
}
```

Example don't

```
void read(int* p, int n);   // read max n integers into *p
```

Example

```
void read(span<int> r); // read into the range of integers r
```

Alternative formulation: Don't postpone to run time what can be done well at compile time.

Enforcement

- Look for pointer arguments.
- Look for run-time checks for range violations.

P.6: What cannot be checked at compile time should be checkable at run time

Reason Leaving hard-to-detect errors in a program is asking for crashes and bad results.

Note Ideally we catch all errors (that are not errors in the programmer's logic) at either compile-time or run-time. It is impossible to catch all errors at compile time and often not affordable to catch all remaining errors at run time. However, we should endeavor to write programs that in principle can be checked, given sufficient resources (analysis programs, run-time checks, machine resources, time).

Example, bad

```
extern void f(int* p);  // separately compiled, possibly dynamically loaded

void g(int n)
{
    f(new int[n]);  // bad: the number of elements is not passed to f()
}
```

Here, a crucial bit of information (the number of elements) has been so thoroughly "obscured" that static analysis is probably rendered infeasible and dynamic checking can be very difficult when `f()` is part of an ABI so that we cannot "instrument" that pointer. We could embed helpful information into the free store, but that requires global changes to a system and maybe to the compiler. What we have here is a design that makes error detection very hard.

Example, bad We can of course pass the number of elements along with the pointer:

```
extern void f2(int* p, int n);  // separately compiled, possibly dynamically loaded

void g2(int n)
{
    f2(new int[n], m);    // bad: the wrong number of elements can be passed to f()
}
```

Passing the number of elements as an argument is better (and far more common) than just passing the pointer and relying on some (unstated) convention for knowing or discovering the number of elements. However (as shown), a simple typo can introduce a serious error. The connection between the two arguments of `f2()` is conventional, rather than explicit.

Also, it is implicit that `f2()` is supposed to `delete` its argument (or did the caller make a second mistake?).

Example, bad The standard library resource management pointers fail to pass the size when they point to an object:

```
extern void f3(unique_ptr<int[]>, int n);    // separately compiled, possibly dynamical
                                             // NB: this assumes the calling code is AB
```

```
                                        // compatible C++ compiler and the same st

void g3(int n)
{
    f3(make_unique<int[]>(n), m);    // bad: pass ownership and size separately
}
```

Example We need to pass the pointer and the number of elements as an integral object:

```
extern void f4(vector<int>&);       // separately compiled, possibly dynamically loaded
extern void f4(span<int>);          // separately compiled, possibly dynamically loaded
                                        // NB: this assumes the calling code is AB
                                        // compatible C++ compiler and the same st

void g3(int n)
{
    vector<int> v(n);
    f4(v);                     // pass a reference, retain ownership
    f4(span<int>{v});    // pass a view, retain ownership
}
```

This design carries the number of elements along as an integral part of an object, so that errors are unlikely and dynamic (run-time) checking is always feasible, if not always affordable.

Example How do we transfer both ownership and all information needed for validating use?

```
vector<int> f5(int n)    // OK: move
{
    vector<int> v(n);
    // ... initialize v ...
    return v;
}

unique_ptr<int[]> f6(int n)    // bad: loses n
{
    auto p = make_unique<int[]>(n);
    // ... initialize *p ...
    return p;
}

owner<int*> f7(int n)    // bad: loses n and we might forget to delete
```

```
{
    owner<int*> p = new int[n];
    // ... initialize *p ...
    return p;
}
```

Example

- ???
- show how possible checks are avoided by interfaces that pass polymorphic base classes around, when they actually know what they need? Or strings as "free-style" options

Enforcement

- Flag (pointer, count)-style interfaces (this will flag a lot of examples that can't be fixed for compatibility reasons)
- ???

P.7: Catch run-time errors early

Reason Avoid "mysterious" crashes. Avoid errors leading to (possibly unrecognized) wrong results.

Example

```
void increment1(int* p, int n)    // bad: error prone
{
    for (int i = 0; i < n; ++i) ++p[i];
}

void use1(int m)
{
    const int n = 10;
    int a[n] = {};
    // ...
    increment1(a, m);   // maybe typo, maybe m <= n is supposed
                        // but assume that m == 20
    // ...
}
```

Here we made a small error in `use1` that will lead to corrupted data or a crash. The (pointer, count)-style interface leaves `increment1()` with no realistic way of defending itself against out-of-range errors. Assuming that we could check subscripts for out of range access, the error would not be discovered until `p[10]` was accessed. We could check earlier and improve the code:

```
void increment2(span<int> p)
{
    for (int& x : p) ++x;
}

void use2(int m)
{
    const int n = 10;
    int a[n] = {};
    // ...
    increment2({a, m});    // maybe typo, maybe m<=n is supposed
    // ...
}
```

Now, `m<=n` can be checked at the point of call (early) rather than later. If all we had was a typo so that we meant to use `n` as the bound, the code could be further simplified (eliminating the possibility of an error):

```
void use3(int m)
{
    const int n = 10;
    int a[n] = {};
    // ...
    increment2(a);   // the number of elements of a need not be repeated
    // ...
}
```

Example, bad Don't repeatedly check the same value. Don't pass structured data as strings:

```
Date read_date(istream& is);    // read date from istream

Date extract_date(const string& s);    // extract date from string

void user1(const string& date)    // manipulate date
{
    auto d = extract_date(date);
    // ...
}
```

```
void user2()
{
    Date d = read_date(cin);
    // ...
    user1(d.to_string());
    // ...
}
```

The date is validated twice (by the `Date` constructor) and passed as a character string (unstructured data).

Example Excess checking can be costly. There are cases where checking early is dumb because you may not ever need the value, or may only need part of the value that is more easily checked than the whole. Similarly, don't add validity checks that change the asymptotic behavior of your interface (e.g., don't add a `O(n)` check to an interface with an average complexity of `O(1)`).

```
class Jet {    // Physics says: e*e < x*x + y*y + z*z

    float x;
    float y;
    float z;
    float e;
public:
    Jet(float x, float y, float z, float e)
        :x(x), y(y), z(z), e(e)
    {
        // Should I check here that the values are physically meaningful?
    }

    float m() const
    {
        // Should I handle the degenerate case here?
        return sqrt(x*x + y*y + z*z - e*e);
    }

    ???
};
```

The physical law for a jet (`e*e < x*x + y*y + z*z`) is not an invariant because of the possibility for measurement errors.

???

Enforcement

- Look at pointers and arrays: Do range-checking early
- Look at conversions: Eliminate or mark narrowing conversions
- Look for unchecked values coming from input
- Look for structured data (objects of classes with invariants) being converted into strings
- ???

P.8: Don't leak any resources

Reason Even a slow growth in resources will, over time, exhaust the availability of those resources. This is particularly important for long-running programs, but is an essential piece of responsible programming behavior.

Example, bad

```
void f(char* name)
{
    FILE* input = fopen(name, "r");
    // ...
    if (something) return;   // bad: if something == true, a file handle is leaked
    // ...
    fclose(input);
}
```

Prefer RAII:

```
void f(char* name)
{
    ifstream input {name};
    // ...
    if (something) return;   // OK: no leak
    // ...
}
```

See also: The resource management section

Note A leak is colloquially "anything that isn't cleaned up." The more important classification is "anything that can no longer be cleaned up." For example, allocating an object on the heap and then losing the last pointer that points to that allocation. This rule should not be taken as requiring that allocations within long-lived objects must be returned during program shutdown. (Although if they can be cleanly and safely de-allocated, they should be.)

Note Enforcing (#In.force)[the lifetime profile] eliminates leaks. When combined with resource safety provided by (#Rr-raii)[RAII], it eliminates the need for "garbage collection" (by generating no garbage). Combine this with enforcement of (the type and bounds profiles)[In.force] and you get complete type- and resource-safety, guaranteed by tools.

Enforcement

- Look at pointers: Classify them into non-owners (the default) and owners. Where feasible, replace owners with standard-library resource handles (as in the example above). Alternatively, mark an owner as such using `owner` from the GSL.
- Look for naked `new` and `delete`
- Look for known resource allocating functions returning raw pointers (such as `fopen`, `malloc`, and `strdup`)

P.9: Don't waste time or space

Reason This is C++.

Note Time and space that you spend well to achieve a goal (e.g., speed of development, resource safety, or simplification of testing) is not wasted. "Another benefit of striving for efficiency is that the process forces you to understand the problem in more depth." - Alex Stepanov

Example, bad

```
struct X {
    char ch;
    int i;
    string s;
    char ch2;

    X& operator=(const X& a);
    X(const X&);
};

X waste(const char* p)
{
    if (p == nullptr) throw Nullptr_error{};
    int n = strlen(p);
    auto buf = new char[n];
    if (buf == nullptr) throw Allocation_error{};
    for (int i = 0; i < n; ++i) buf[i] = p[i];
```

```
    // ... manipulate buffer ...
    X x;
    x.ch = 'a';
    x.s = string(n);    // give x.s space for *ps
    for (int i = 0; i < x.s.size(); ++i) x.s[i] = buf[i];  // copy buf into x.s
    delete buf;
    return x;
}

void driver()
{
    X x = waste("Typical argument");
    // ...
}
```

Yes, this is a caricature, but we have seen every individual mistake in production code, and worse. Note that the layout of `X` guarantees that at least 6 bytes (and most likely more) bytes are wasted. The spurious definition of copy operations disables move semantics so that the return operation is slow. The use of `new` and `delete` for `buf` is redundant; if we really needed a local string, we should use a local `string`. There are several more performance bugs and gratuitous complication.

Example, bad

```
void lower(zstring s)
{
    for (int i = 0; i<strlen(s); ++s) s[i] = tolower(s[i]);
}
```

Yes, this is an example from production code. We leave it to the reader to figure out what's wasted.

Note An individual example of waste is rarely significant, and where it is significant, it is typically easily eliminated by an expert. However, waste spread liberally across a code base can easily be significant and experts are not always as available as we would like. The aim of this rule (and the more specific rules that support it) is to eliminate most waste related to the use of C++ before it happens. After that, we can look at waste related to algorithms and requirements, but that is beyond the scope of these guidelines.

Enforcement Many more specific rules aim at the overall goals of simplicity and elimination of gratuitous waste.

P.10: Prefer immutable data to mutable data

Reason It is easier to reason about constants than about variables. Sumething immutable cannot change unexpectedly. Sometimes immutability enables better optimization. You can't have a data race on a constant.

See Con: Constants and Immutability

I: Interfaces

An interface is a contract between two parts of a program. Precisely stating what is expected of a supplier of a service and a user of that service is essential. Having good (easy-to-understand, encouraging efficient use, not error-prone, supporting testing, etc.) interfaces is probably the most important single aspect of code organization.

Interface rule summary:

- I.1: Make interfaces explicit
- I.2: Avoid global variables
- I.3: Avoid singletons
- I.4: Make interfaces precisely and strongly typed
- I.5: State preconditions (if any)
- I.6: Prefer `Expects()` for expressing preconditions
- I.7: State postconditions
- I.8: Prefer `Ensures()` for expressing postconditions
- I.9: If an interface is a template, document its parameters using concepts
- I.10: Use exceptions to signal a failure to perform a required tasks
- I.11: Never transfer ownership by a raw pointer (`T*`)
- I.12: Declare a pointer that must not be null as `not_null`
- I.13: Do not pass an array as a single pointer
- I.22: Avoid complex initialization of global objects
- I.23: Keep the number of function arguments low
- I.24: Avoid adjacent unrelated parameters of the same type
- I.25: Prefer abstract classes as interfaces to class hierarchies
- I.26: If you want a cross-compiler ABI, use a C-style subset

See also

- F: Functions
- C.concrete: Concrete types
- C.hier: Class hierarchies
- C.over: Overloading and overloaded operators
- C.con: Containers and other resource handles

- E: Error handling
- T: Templates and generic programming

I.1: Make interfaces explicit

Reason Correctness. Assumptions not stated in an interface are easily overlooked and hard to test.

Example, bad Controlling the behavior of a function through a global (namespace scope) variable (a call mode) is implicit and potentially confusing. For example:

```
int rnd(double d)
{
    return (rnd_up) ? ceil(d) : d;    // don't: "invisible" dependency
}
```

It will not be obvious to a caller that the meaning of two calls of `rnd(7.2)` might give different results.

Exception: Sometimes we control the details of a set of operations by an environment variable, e.g., normal vs. verbose output or debug vs. optimized. The use of a non-local control is potentially confusing, but controls only implementation details of otherwise fixed semantics.

Example, bad Reporting through non-local variables (e.g., `errno`) is easily ignored. For example:

```
fprintf(connection, "logging: %d %d %d\n", x, y, s); // don't: no test of printf's retu
```

What if the connection goes down so that no logging output is produced? See I.??.

Alternative: Throw an exception. An exception cannot be ignored.

Alternative formulation: Avoid passing information across an interface through non-local or implicit state. Note that non-`const` member functions pass information to other member functions through their object's state.

Alternative formulation: An interface should be a function or a set of functions. Functions can be template functions and sets of functions can be classes or class templates.

Enforcement

- (Simple) A function should not make control-flow decisions based on the values of variables declared at namespace scope.
- (Simple) A function should not write to variables declared at namespace scope.

I.2 Avoid global variables

Reason Non-`const` global variables hide dependencies and make the dependencies subject to unpredictable changes.

Example

```
struct Data {
    // ... lots of stuff ...
} data;            // non-const data

void compute()     // don't
{
    // ... use data ...
}

void output()     // don't
{
    // ... use data ...
}
```

Who else might modify `data`?

Note Global constants are useful.

Note The rule against global variables applies to namespace scope variables as well.

Alternative: If you use global (more generally namespace scope data) to avoid copying, consider passing the data as an object by reference to const. Another solution is to define the data as the state of some object and the operations as member functions.

Warning: Beware of data races: If one thread can access nonlocal data (or data passed by reference) while another thread executes the callee, we can have a data race. Every pointer or reference to mutable data is a potential data race.

Note You cannot have a race condition on immutable data.

Reference: See the rules for calling functions.

Enforcement (Simple) Report all non-`const` variables declared at namespace scope.

I.3: Avoid singletons

Reason Singletons are basically complicated global objects in disguise.

Example

```
class Singleton {
    // ... lots of stuff to ensure that only one Singleton object is created,
    // that it is initialized properly, etc.
};
```

There are many variants of the singleton idea. That's part of the problem.

Note If you don't want a global object to change, declare it `const` or `constexpr`.

Exception You can use the simplest "singleton" (so simple that it is often not considered a singleton) to get initialization on first use, if any:

```
X& myX()
{
    static X my_x {3};
    return my_x;
}
```

This is one of the most effective solutions to problems related to initialization order. In a multi-threaded environment the initialization of the static object does not introduce a race condition (unless you carelessly access a shared object from within its constructor).

If you, as many do, define a singleton as a class for which only one object is created, functions like `myX` are not singletons, and this useful technique is not an exception to the no-singleton rule.

Enforcement Very hard in general.

- Look for classes with names that include `singleton`.
- Look for classes for which only a single object is created (by counting objects or by examining constructors).
- If a class X has a public static function that contains a function-local static of the class' type X and returns a pointer or reference to it, ban that.

I.4: Make interfaces precisely and strongly typed

Reason Types are the simplest and best documentation, have well-defined meaning, and are guaranteed to be checked at compile time. Also, precisely typed code is often optimized better.

Example, don't Consider:

```
void pass(void* data);    // void* is suspicious
```

Now the callee has to cast the data pointer (back) to a correct type to use it. That is error-prone and often verbose. Avoid `void*`, especially in interfaces. Consider using a `variant` or a pointer to base instead. (Future note: Consider a pointer to concept.)

Alternative: Often, a template parameter can eliminate the `void*` turning it into a `T*` or `T&`.

Example, bad Consider:

```
void draw_rect(int, int, int, int);   // great opportunities for mistakes

draw_rect(p.x, p.y, 10, 20);          // what does 10, 20 mean?
```

An `int` can carry arbitrary forms of information, so we must guess about the meaning of the four `int`s. Most likely, the first two are an `x`,`y` coordinate pair, but what are the last two? Comments and parameter names can help, but we could be explicit:

```
void draw_rectangle(Point top_left, Point bottom_right);
void draw_rectangle(Point top_left, Size height_width);

draw_rectangle(p, Point{10, 20}); // two corners
draw_rectangle(p, Size{10, 20});  // one corner and a (height, width) pair
```

Obviously, we cannot catch all errors through the static type system (e.g., the fact that a first argument is supposed to be a top-left point is left to convention (naming and comments)).

Example, bad In the following example, it is not clear from the interface what `time_to_blink` means: Seconds? Milliseconds?

```
void blink_led(int time_to_blink) // bad - the unit is ambiguous
{
    // ...
    // do something with time_to_blink
    // ...
}

void use()
{
    blink_led(2);
}
```

Example, good `std::chrono::duration` types introduced in C++11 helps making the unit of time duration explicit.

```
void blink_led(milliseconds time_to_blink) // good - the unit is explicit
{
    // ...
    // do something with time_to_blink
    // ...
}

void use()
{
    blink_led(1500ms);
}
```

The function can also be written in such a way that it will accept any time duration unit.

```
template<class rep, class period>
void blink_led(duration<rep, period> time_to_blink) // good - accepts any unit
{
    // assuming that millisecond is the smallest relevant unit
    auto milliseconds_to_blink = duration_cast<milliseconds>(time_to_blink);
    // ...
    // do something with milliseconds_to_blink
    // ...
}

void use()
{
    blink_led(2s);
    blink_led(1500ms);
}
```

Enforcement

- (Simple) Report the use of `void*` as a parameter or return type.
- (Hard to do well) Look for member functions with many built-in type arguments.

I.5: State preconditions (if any)

Reason Arguments have meaning that may constrain their proper use in the callee.

Example Consider:

```
double sqrt(double x);
```

Here `x` must be nonnegative. The type system cannot (easily and naturally) express that, so we must use other means. For example:

```
double sqrt(double x); // x must be nonnegative
```

Some preconditions can be expressed as assertions. For example:

```
double sqrt(double x) { Expects(x >= 0); /* ... */ }
```

Ideally, that `Expects(x >= 0)` should be part of the interface of `sqrt()` but that's not easily done. For now, we place it in the definition (function body).

Reference: `Expects()` is described in GSL.

Note Prefer a formal specification of requirements, such as `Expects(p != nullptr);`. If that is infeasible, use English text in comments, such as `// the sequence [p:q) is ordered using <`.

Note Most member functions have as a precondition that some class invariant holds. That invariant is established by a constructor and must be reestablished upon exit by every member function called from outside the class. We don't need to mention it for each member function.

Enforcement (Not enforceable)

See also: The rules for passing pointers. ???

I.6: Prefer `Expects()` for expressing preconditions

Reason To make it clear that the condition is a precondition and to enable tool use.

Example

```
int area(int height, int width)
{
    Expects(height > 0 && width > 0);            // good
    if (height <= 0 || width <= 0) my_error();   // obscure
    // ...
}
```

Note Preconditions can be stated in many ways, including comments, `if`-statements, and `assert()`. This can make them hard to distinguish from ordinary code, hard to update, hard to manipulate by tools, and may have the wrong semantics (do you always want to abort in debug mode and check nothing in productions runs?).

Note Preconditions should be part of the interface rather than part of the implementation, but we don't yet have the language facilities to do that.

Note `Expects()` can also be used to check a condition in the middle of an algorithm.

Enforcement (Not enforceable) Finding the variety of ways preconditions can be asserted is not feasible. Warning about those that can be easily identified (`assert()`) has questionable value in the absence of a language facility.

I.7: State postconditions

Reason To detect misunderstandings about the result and possibly catch erroneous implementations.

Example, bad Consider:

```
int area(int height, int width) { return height * width; }  // bad
```

Here, we (incautiously) left out the precondition specification, so it is not explicit that height and width must be positive. We also left out the postcondition specification, so it is not obvious that the algorithm (`height * width`) is wrong for areas larger than the largest integer. Overflow can happen. Consider using:

```
int area(int height, int width)
{
    auto res = height * width;
    Ensures(res > 0);
    return res;
}
```

Example, bad Consider a famous security bug:

```
void f()    // problematic
{
    char buffer[MAX];
    // ...
    memset(buffer, 0, MAX);
}
```

There was no postcondition stating that the buffer should be cleared and the optimizer eliminated the apparently redundant `memset()` call:

```
void f()    // better
{
    char buffer[MAX];
    // ...
    memset(buffer, 0, MAX);
    Ensures(buffer[0] == 0);
}
```

Note Postconditions are often informally stated in a comment that states the purpose of a function; `Ensures()` can be used to make this more systematic, visible, and checkable.

Note Postconditions are especially important when they relate to something that is not directly reflected in a returned result, such as a state of a data structure used.

Example Consider a function that manipulates a `Record`, using a `mutex` to avoid race conditions:

```
mutex m;

void manipulate(Record& r)    // don't
{
    m.lock();
    // ... no m.unlock() ...
}
```

Here, we "forgot" to state that the `mutex` should be released, so we don't know if the failure to ensure release of the `mutex` was a bug or a feature. Stating the postcondition would have made it clear:

```
void manipulate(Record& r)    // postcondition: m is unlocked upon exit
{
    m.lock();
    // ... no m.unlock() ...
}
```

The bug is now obvious (but only to a human reading comments)

Better still, use RAII to ensure that the postcondition ("the lock must be released") is enforced in code:

```
void manipulate(Record& r)    // best
{
    lock_guard<mutex> _ {m};
    // ...
}
```

Note Ideally, postconditions are stated in the interface/declaration so that users can easily see them. Only postconditions related to the users can be stated in the interface. Postconditions related only to internal state belongs in the definition/implementation.

Enforcement (Not enforceable) This is a philosophical guideline that is infeasible to check directly in the general case. Domain specific checkers (like lock-holding checkers) exist for many toolchains.

I.8: Prefer `Ensures()` for expressing postconditions

Reason To make it clear that the condition is a postcondition and to enable tool use.

Example

```
void f()
{
    char buffer[MAX];
    // ...
    memset(buffer, 0, MAX);
    Ensures(buffer[0] == 0);
}
```

Note Postconditions can be stated in many ways, including comments, `if`-statements, and `assert()`. This can make them hard to distinguish from ordinary code, hard to update, hard to manipulate by tools, and may have the wrong semantics.

Alternative: Postconditions of the form "this resource must be released" are best expressed by RAII.

Ideally, that `Ensures` should be part of the interface, but that's not easily done. For now, we place it in the definition (function body).

Enforcement (Not enforceable) Finding the variety of ways postconditions can be asserted is not feasible. Warning about those that can be easily identified (`assert()`) has questionable value in the absence of a language facility.

I.9: If an interface is a template, document its parameters using concepts

Reason Make the interface precisely specified and compile-time checkable in the (not so distant) future.

Example Use the ISO Concepts TS style of requirements specification. For example:

```
template<typename Iter, typename Val>
// requires InputIterator<Iter> && EqualityComparable<ValueType<Iter>>, Val>
Iter find(Iter first, Iter last, Val v)
{
    // ...
}
```

Note Soon (maybe in 2016), most compilers will be able to check `requires` clauses once the `//` is removed.

See also: See generic programming and concepts.

Enforcement (Not enforceable yet) A language facility is under specification. When the language facility is available, warn if any non-variadic template parameter is not constrained by a concept (in its declaration or mentioned in a `requires` clause).

I.10: Use exceptions to signal a failure to perform a required task

Reason It should not be possible to ignore an error because that could leave the system or a computation in an undefined (or unexpected) state. This is a major source of errors.

Example

```
int printf(const char* ...);    // bad: return negative number if output fails

template <class F, class ...Args>
explicit thread(F&& f, Args&&... args);    // good: throw system_error if unable to sta
```

Note: What is an error? An error means that the function cannot achieve its advertised purpose (including establishing postconditions). Calling code that ignores the error could lead to wrong results or undefined systems state. For example, not being able to connect to a remote server is not by itself an error: the server can refuse a connection for all kinds of reasons, so the natural thing is to return a result that the caller always has to check. However, if failing to make a connection is considered an error, then a failure should throw an exception.

Exception: Many traditional interface functions (e.g., UNIX signal handlers) use error codes (e.g., `errno`) to report what are really status codes, rather than errors. You don't have a good alternative to using such, so calling these does not violate the rule.

Alternative: If you can't use exceptions (e.g. because your code is full of old-style raw-pointer use or because there are hard-real-time constraints), consider using a style that returns a pair of values:

```
int val;
int error_code;
tie(val, error_code) = do_something();
if (error_code == 0) {
    // ... handle the error or exit ...
}
// ... use val ...
```

Note We don't consider "performance" a valid reason not to use exceptions.

- Often, explicit error checking and handling consume as much time and space as exception handling.
- Often, cleaner code yields better performance with exceptions (simplifying the tracing of paths through the program and their optimization).
- A good rule for performance critical code is to move checking outside the critical part of the code (checking).
- In the longer term, more regular code gets better optimized.

See also: I.5 and I.7 for reporting precondition and postcondition violations.

Enforcement

- (Not enforceable) This is a philosophical guideline that is infeasible to check directly.
- Look for `errno`.

I.11: Never transfer ownership by a raw pointer (`T*`)

Reason If there is any doubt whether the caller or the callee owns an object, leaks or premature destruction will occur.

Example Consider:

```
X* compute(args)    // don't
{
    X* res = new X{};
    // ...
    return res;
}
```

Who deletes the returned `X`? The problem would be harder to spot if compute returned a reference. Consider returning the result by value (use move semantics if the result is large):

```
vector<double> compute(args)  // good
{
    vector<double> res(10000);
    // ...
    return res;
}
```

Alternative: Pass ownership using a "smart pointer", such as `unique_ptr` (for exclusive ownership) and `shared_ptr` (for shared ownership). However that is less elegant and less efficient unless reference semantics are needed.

Alternative: Sometimes older code can't be modified because of ABI compatibility requirements or lack of resources. In that case, mark owning pointers using `owner`:

```
owner<X*> compute(args)    // It is now clear that ownership is transferred
{
    owner<X*> res = new X{};
    // ...
    return res;
}
```

This tells analysis tools that `res` is an owner. That is, its value must be `deleted` or transferred to another owner, as is done here by the `return`.

`owner` is used similarly in the implementation of resource handles.

`owner` is defined in the guideline support library.

Note Every object passed as a raw pointer (or iterator) is assumed to be owned by the caller, so that its lifetime is handled by the caller. Viewed another way: ownership transferring APIs are relatively rare compared to pointer-passing APIs, so the default is "no ownership transfer."

See also: Argument passing and value return.

Enforcement

- (Simple) Warn on `delete` of a raw pointer that is not an `owner`.
- (Simple) Warn on failure to either `reset` or explicitly `delete` an `owner` pointer on every code path.
- (Simple) Warn if the return value of `new` or a function call with return value of pointer type is assigned to a raw pointer.

I.12: Declare a pointer that must not be null as `not_null`

Reason To help avoid dereferencing `nullptr` errors. To improve performance by avoiding redundant checks for `nullptr`.

Example

```
int length(const char* p);            // it is not clear whether length(nullptr) is val

length(nullptr);                      // OK?

int length(not_null<const char*> p);  // better: we can assume that p cannot be nullptr

int length(const char* p);            // we must assume that p can be nullptr
```

By stating the intent in source, implementers and tools can provide better diagnostics, such as finding some classes of errors through static analysis, and perform optimizations, such as removing branches and null tests.

Note The assumption that the pointer to `char` pointed to a C-style string (a zero-terminated string of characters) was still implicit, and a potential source of confusion and errors. Use `zstring` in preference to `const char*`.

```
int length(not_null<zstring> p);    // we can assume that p cannot be nullptr
                                    // we can assume that p points to a zero-terminated
```

Note: `length()` is, of course, `std::strlen()` in disguise.

Enforcement

- (Simple) ((Foundation)) If a function checks a pointer parameter against `nullptr` before access, on all control-flow paths, then warn it should be declared `not_null`.
- (Complex) If a function with pointer return value ensures it is not `nullptr` on all return paths, then warn the return type should be declared `not_null`.

I.13: Do not pass an array as a single pointer

Reason (pointer, size)-style interfaces are error-prone. Also, a plain pointer (to array) must rely on some convention to allow the callee to determine the size.

Example Consider:

```
void copy_n(const T* p, T* q, int n); // copy from [p:p+n) to [q:q+n)
```

What if there are fewer than `n` elements in the array pointed to by `q`? Then, we overwrite some probably unrelated memory. What if there are fewer than `n` elements in the array pointed to by `p`? Then, we read some probably unrelated memory. Either is undefined behavior and a potentially very nasty bug.

Alternative Consider using explicit spans:

```
void copy(span<const T> r, span<T> r2); // copy r to r2
```

Example, bad Consider:

```
void draw(Shape* p, int n);  // poor interface; poor code
Circle arr[10];
// ...
draw(arr, 10);
```

Passing `10` as the `n` argument may be a mistake: the most common convention is to assume [`0`:`n`) but that is nowhere stated. Worse is that the call of `draw()` compiled at all: there was an implicit conversion from array to pointer (array decay) and then another implicit conversion from `Circle` to `Shape`. There is no way that `draw()` can safely iterate through that array: it has no way of knowing the size of the elements.

Alternative: Use a support class that ensures that the number of elements is correct and prevents dangerous implicit conversions. For example:

```
void draw2(span<Circle>);
Circle arr[10];
// ...
draw2(span<Circle>(arr));  // deduce the number of elements
draw2(arr);    // deduce the element type and array size

void draw3(span<Shape>);
draw3(arr);    // error: cannot convert Circle[10] to span<Shape>
```

This `draw2()` passes the same amount of information to `draw()`, but makes the fact that it is supposed to be a range of `Circle`s explicit. See ???.

Exception: Use `zstring` and `czstring` to represent a C-style, zero-terminated strings. But when doing so, use `string_span` from the (GSL)[#GSL] to prevent range errors.

Enforcement

- (Simple) ((Bounds)) Warn for any expression that would rely on implicit conversion of an array type to a pointer type. Allow exception for zstring/czstring pointer types.
- (Simple) ((Bounds)) Warn for any arithmetic operation on an expression of pointer type that results in a value of pointer type. Allow exception for zstring/czstring pointer types.

I.22: Avoid complex initialization of global objects

Reason Complex initialization can lead to undefined order of execution.

Example

```
// file1.c

extern const X x;

const Y y = f(x);   // read x; write y

// file2.c

extern const Y y;

const X x = g(y);   // read y; write x
```

Since `x` and `y` are in different translation units the order of calls to `f()` and `g()` is undefined; one will access an uninitialized `const`. This particular example shows that the order-of-initialization problem for global (namespace scope) objects is not limited to global *variables*.

Note Order of initialization problems become particularly difficult to handle in concurrent code. It is usually best to avoid global (namespace scope) objects altogether.

Enforcement

- Flag initializers of globals that call non-`constexpr` functions
- Flag initializers of globals that access `extern` objects

I.23: Keep the number of function arguments low

Reason Having many arguments opens opportunities for confusion. Passing lots of arguments is often costly compared to alternatives.

Example The standard-library `merge()` is at the limit of what we can comfortably handle

```
template<class InputIterator1, class InputIterator2, class OutputIterator, class Compare
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result, Compare comp);
```

Here, we have four template arguments and six function arguments. To simplify the most frequent and simplest uses, the comparison argument can be defaulted to `<`:

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result);
```

This doesn't reduce the total complexity, but it reduces the surface complexity presented to many users. To really reduce the number of arguments, we need to bundle the arguments into higher-level abstractions:

```
template<class InputRange1, class InputRange2, class OutputIterator>
OutputIterator merge(InputRange1 r1, InputRange2 r2, OutputIterator result);
```

Grouping arguments into "bundles" is a general technique to reduce the number of arguments and to increase the opportunities for checking.

Note How many arguments are too many? Four arguments is a lot. There are functions that are best expressed with four individual arguments, but not many.

Alternative: Group arguments into meaningful objects and pass the objects (by value or by reference).

Alternative: Use default arguments or overloads to allow the most common forms of calls to be done with fewer arguments.

Enforcement

- Warn when a functions declares two iterators (including pointers) of the same type instead of a range or a view.
- (Not enforceable) This is a philosophical guideline that is infeasible to check directly.

I.24: Avoid adjacent unrelated parameters of the same type

Reason Adjacent arguments of the same type are easily swapped by mistake.

Example, bad Consider:

```
void copy_n(T* p, T* q, int n);  // copy from [p:p+n) to [q:q+n)
```

This is a nasty variant of a K&R C-style interface. It is easy to reverse the "to" and "from" arguments.

Use `const` for the "from" argument:

```
void copy_n(const T* p, T* q, int n);  // copy from [p:p+n) to [q:q+n)
```

Example If the order of the parameters is not important, there is no problem:

```
int max(int a, int b);
```

Alternative Don't pass arrays as pointers, pass an object representing a range (e.g., a `span`):

```
void copy_n(span<const T> p, span<T> q);  // copy from p to q
```

Alternative Define a struct as the parameter type and name the fields for those parameters accordingly:

```
struct SystemParams {
    string config_file;
    string output_path;
    seconds timeout;
};
void initialize(SystemParams p);
```

This has a tendency to make invocations of this clear to future readers, as the parameters are often filled in by name at the call site.

Enforcement (Simple) Warn if two consecutive parameters share the same type.

I.25: Prefer abstract classes as interfaces to class hierarchies

Reason Abstract classes are more likely to be stable than base classes with state.

Example, bad You just knew that `Shape` would turn up somewhere :-)

```
class Shape {  // bad: interface class loaded with data
public:
    Point center() const { return c; }
    virtual void draw() const;
    virtual void rotate(int);
    // ...
private:
    Point c;
    vector<Point> outline;
    Color col;
};
```

This will force every derived class to compute a center – even if that's non-trivial and the center is never used. Similarly, not every `Shape` has a `Color`, and many `Shape`s are best represented without an outline defined as a sequence of `Point`s. Abstract classes were invented to discourage users from writing such classes:

```
class Shape {    // better: Shape is a pure interface
public:
    virtual Point center() const = 0;   // pure virtual function
    virtual void draw() const = 0;
    virtual void rotate(int) = 0;
    // ...
    // ... no data members ...
};
```

Enforcement (Simple) Warn if a pointer to a class `C` is assigned to a pointer to a base of `C` and the base class contains data members.

I.26: If you want a cross-compiler ABI, use a C-style subset

Reason Different compilers implement different binary layouts for classes, exception handling, function names, and other implementation details.

Exception: You can carefully craft an interface using a few carefully selected higher-level C++ types. See ???.

Exception: Common ABIs are emerging on some platforms freeing you from the more draconian restrictions.

Note If you use a single compiler, you can use full C++ in interfaces. That may require recompilation after an upgrade to a new compiler version.

Enforcement (Not enforceable) It is difficult to reliably identify where an interface forms part of an ABI.

F: Functions

A function specifies an action or a computation that takes the system from one consistent state to the next. It is the fundamental building block of programs.

It should be possible to name a function meaningfully, to specify the requirements of its argument, and clearly state the relationship between the arguments and the result. An implementation is not a specification. Try to think about what a function does as well as about how it does it. Functions are the most critical part in most interfaces, so see the interface rules.

Function rule summary:

Function definition rules:

- F.1: "Package" meaningful operations as carefully named functions
- F.2: A function should perform a single logical operation
- F.3: Keep functions short and simple
- F.4: If a function may have to be evaluated at compile time, declare it `constexpr`
- F.5: If a function is very small and time-critical, declare it inline
- F.6: If your function may not throw, declare it `noexcept`
- F.7: For general use, take `T*` or `T&` arguments rather than smart pointers
- F.8: Prefer pure functions

Parameter passing expression rules:

- F.15: Prefer simple and conventional ways of passing information
- F.16: For "in" parameters, pass cheaply-copied types by value and others by reference to `const`
- F.17: For "in-out" parameters, pass by reference to non-`const`
- F.18: For "consume" parameters, pass by `X&&` and `std::move` the parameter
- F.19: For "forward" parameters, pass by `TP&&` and only `std::forward` the parameter
- F.20: For "out" output values, prefer return values to output parameters
- F.21: To return multiple "out" values, prefer returning a tuple or struct
- F.60: Prefer `T*` over `T&` when "no argument" is a valid option

Parameter passing semantic rules:

- F.22: Use `T*` or `owner<T*>` or a smart pointer to designate a single object
- F.23: Use a `not_null<T>` to indicate "null" is not a valid value

- F.24: Use a `span<T>` or a `span_p<T>` to designate a half-open sequence
- F.25: Use a `zstring` or a `not_null<zstring>` to designate a C-style string
- F.26: Use a `unique_ptr<T>` to transfer ownership where a pointer is needed
- F.27: Use a `shared_ptr<T>` to share ownership

Value return semantic rules:

- F.42: Return a `T*` to indicate a position (only)
- F.43: Never (directly or indirectly) return a pointer to a local object
- F.44: Return a `T&` when copy is undesirable and "returning no object" isn't an option
- F.45: Don't return a `T&&`
- F.46: `int` is the return type for `main()`
- F.47: Return `T&` from assignment operators.

Other function rules:

- F.50: Use a lambda when a function won't do (to capture local variables, or to write a local function)
- F.51: Where there is a choice, prefer default arguments over overloading
- F.52: Prefer capturing by reference in lambdas that will be used locally, including passed to algorithms
- F.53: Avoid capturing by reference in lambdas that will be used nonlocally, including returned, stored on the heap, or passed to another thread
- F.54: If you capture `this`, capture all variables explicitly (no default capture)

Functions have strong similarities to lambdas and function objects so see also Section ???.

F.def: Function definitions

A function definition is a function declaration that also specifies the function's implementation, the function body.

F.1: "Package" meaningful operations as carefully named functions

Reason Factoring out common code makes code more readable, more likely to be reused, and limit errors from complex code. If something is a well-specified action, separate it out from its surrounding code and give it a name.

Example, don't

```
void read_and_print(istream& is)    // read and print an int
{
    int x;
    if (is >> x)
        cout << "the int is " << x << '\n';
    else
        cerr << "no int on input\n";
}
```

Almost everything is wrong with `read_and_print`. It reads, it writes (to a fixed `ostream`), it writes error messages (to a fixed `ostream`), it handles only `int`s. There is nothing to reuse, logically separate operations are intermingled and local variables are in scope after the end of their logical use. For a tiny example, this looks OK, but if the input operation, the output operation, and the error handling had been more complicated the tangled mess could become hard to understand.

Note If you write a non-trivial lambda that potentially can be used in more than one place, give it a name by assigning it to a (usually non-local) variable.

Example

```
sort(a, b, [](T x, T y) { return x.rank() < y.rank() && x.value() < y.value(); });
```

Naming that lambda breaks up the expression into its logical parts and provides a strong hint to the meaning of the lambda.

```
auto lessT = [](T x, T y) { return x.rank() < y.rank() && x.value() < y.value(); };

sort(a, b, lessT);
find_if(a, b, lessT);
```

The shortest code is not always the best for performance or maintainability.

Exception: Loop bodies, including lambdas used as loop bodies, rarely need to be named. However, large loop bodies (e.g., dozens of lines or dozens of pages) can be a problem. The rule Keep functions short implies "Keep loop bodies short." Similarly, lambdas used as callback arguments are sometimes non-trivial, yet unlikely to be re-usable.

Enforcement

- See Keep functions short
- Flag identical and very similar lambdas used in different places.

F.2: A function should perform a single logical operation

Reason A function that performs a single operation is simpler to understand, test, and reuse.

Example Consider:

```
void read_and_print()    // bad
{
    int x;
    cin >> x;
    // check for errors
    cout << x << "\n";
}
```

This is a monolith that is tied to a specific input and will never find a another (different) use. Instead, break functions up into suitable logical parts and parameterize:

```
int read(istream& is)    // better
{
    int x;
    is >> x;
    // check for errors
    return x;
}

void print(ostream& os, int x)
{
    os << x << "\n";
}
```

These can now be combined where needed:

```
void read_and_print()
{
    auto x = read(cin);
    print(cout, x);
}
```

If there was a need, we could further templatize `read()` and `print()` on the data type, the I/O mechanism, the response to errors, etc. Example:

```
auto read = [](auto& input, auto& value)    // better
{
    input >> value;
    // check for errors
};

auto print(auto& output, const auto& value)
{
    output << value << "\n";
}
```

Enforcement

- Consider functions with more than one "out" parameter suspicious. Use return values instead, including `tuple` for multiple return values.
- Consider "large" functions that don't fit on one editor screen suspicious. Consider factoring such a function into smaller well-named suboperations.
- Consider functions with 7 or more parameters suspicious.

F.3: Keep functions short and simple

Reason Large functions are hard to read, more likely to contain complex code, and more likely to have variables in larger than minimal scopes. Functions with complex control structures are more likely to be long and more likely to hide logical errors

Example Consider:

```
double simpleFunc(double val, int flag1, int flag2)
    // simpleFunc: takes a value and calculates the expected ASIC output, given the tw
{

    double intermediate;
    if (flag1 > 0) {
        intermediate = func1(val);
        if (flag2 % 2)
             intermediate = sqrt(intermediate);
    }
    else if (flag1 == -1) {
        intermediate = func1(-val);
        if (flag2 % 2)
             intermediate = sqrt(-intermediate);
        flag1 = -flag1;
    }
```

```
    if (abs(flag2) > 10) {
        intermediate = func2(intermediate);
    }
    switch (flag2 / 10) {
        case 1: if (flag1 == -1) return finalize(intermediate, 1.171); break;
        case 2: return finalize(intermediate, 13.1);
        default: ;
    }
    return finalize(intermediate, 0.);
}
```

This is too complex (and also pretty long). How would you know if all possible alternatives have been correctly handled? Yes, it breaks other rules also.

We can refactor:

```
double func1_muon(double val, int flag)
{
    // ???
}

double funct1_tau(double val, int flag1, int flag2)
{
    // ???
}

double simpleFunc(double val, int flag1, int flag2)
    // simpleFunc: takes a value and calculates the expected ASIC output, given the tw
{
    if (flag1 > 0)
        return func1_muon(val, flag2);
    if (flag1 == -1)
        return func1_tau(-val, flag1, flag2);    // handled by func1_tau: flag1 = -flag
    return 0.;
}
```

Note “It doesn’t fit on a screen” is often a good practical definition of “far too large.” One-to-five-line functions should be considered normal.

Note Break large functions up into smaller cohesive and named functions. Small simple functions are easily inlined where the cost of a function call is significant.

Enforcement

- Flag functions that do not "fit on a screen." How big is a screen? Try 60 lines by 140 characters; that's roughly the maximum that's comfortable for a book page.
- Flag functions that are too complex. How complex is too complex? You could use cyclomatic complexity. Try "more than 10 logical path through." Count a simple switch as one path.

F.4: If a function may have to be evaluated at compile time, declare it `constexpr`

Reason `constexpr` is needed to tell the compiler to allow compile-time evaluation.

Example The (in)famous factorial:

```
constexpr int fac(int n)
{
    constexpr int max_exp = 17;      // constexpr enables this to be used in Expects
    Expects(0 <= n && n < max_exp);  // prevent silliness and overflow
    int x = 1;
    for (int i = 2; i <= n; ++i) x *= i;
    return x;
}
```

This is C++14. For C++11, use a recursive formulation of `fac()`.

Note `constexpr` does not guarantee compile-time evaluation; it just guarantees that the function can be evaluated at compile time for constant expression arguments if the programmer requires it or the compiler decides to do so to optimize.

```
constexpr int min(int x, int y) { return x < y ? x : y; }

void test(int v)
{
    int m1 = min(-1, 2);            // probably compile-time evaluation
    constexpr int m2 = min(-1, 2);  // compile-time evaluation
    int m3 = min(-1, v);            // run-time evaluation
    constexpr int m4 = min(-1, v);  // error: cannot evaluate at compile-time
}
```

Note `constexpr` functions are pure: they can have no side effects.

```
int dcount = 0;
constexpr int double(int v)
{
    ++dcount;   // error: attempted side effect from constexpr function
    return v + v;
}
```

This is usually a very good thing.

Note Don't try to make all functions `constexpr`. Most computation is best done at run time.

Note Any API that may eventually depend on high-level runtime configuration or business logic should not be made `constexpr`. Such customization can not be evaluated by the compiler, and any `constexpr` functions that depend upon that API will have to be refactored or drop `constexpr`.

Enforcement Impossible and unnecessary. The compiler gives an error if a non-`constexpr` function is called where a constant is required.

F.5: If a function is very small and time-critical, declare it `inline`

Reason Some optimizers are good at inlining without hints from the programmer, but don't rely on it. Measure! Over the last 40 years or so, we have been promised compilers that can inline better than humans without hints from humans. We are still waiting. Specifying `inline` encourages the compiler to do a better job.

Example

```
???
```

Exception: Do not put an `inline` function in what is meant to be a stable interface unless you are really sure that it will not change. An inline function is part of the ABI.

Note `constexpr` implies `inline`.

Note Member functions defined in-class are `inline` by default.

Exception: Template functions (incl. template member functions) must be in headers and therefore inline.

Enforcement Flag `inline` functions that are more than three statements and could have been declared out of line (such as class member functions). To fix: Declare the function out of line. (NM: Certainly possible, but size-based metrics can be very annoying.)

F.6: If your function may not throw, declare it `noexcept`

Reason If an exception is not supposed to be thrown, the program cannot be assumed to cope with the error and should be terminated as soon as possible. Declaring a function `noexcept` helps optimizers by reducing the number of alternative execution paths. It also speeds up the exit after failure.

Example Put `noexcept` on every function written completely in C or in any other language without exceptions. The C++ standard library does that implicitly for all functions in the C standard library.

Note `constexpr` functions cannot throw, so you don't need to use `noexcept` for those.

Example You can use `noexcept` even on functions that can throw:

```
vector<string> collect(istream& is) noexcept
{
    vector<string> res;
    for (string s; is >> s;)
        res.push_back(s);
    return res;
}
```

If `collect()` runs out of memory, the program crashes. Unless the program is crafted to survive memory exhaustion, that may be just the right thing to do; `terminate()` may generate suitable error log information (but after memory runs out it is hard to do anything clever).

Note You must be aware of the execution environment that your code is running when deciding whether to tag a function `noexcept`, especially because of the issue of throwing and allocation. Code that is intended to be perfectly general (like the standard library and other utility code of that sort) needs to support environments where a `bad_alloc` exception may be handled meaningfully. However, the majority of programs and execution environments cannot meaningfully handle a failure to allocate, and aborting the program is the cleanest and simplest response to an allocation failure in those cases. If you know that your application code cannot respond to an allocation failure, it may be appropriate to add `noexcept` even on functions that allocate.

Put another way: In most programs, most functions can throw (e.g., because they use `new`, call functions that do, or use library functions that reports failure by throwing), so don't just sprinkle `noexcept` all over the place without considering whether the possible exceptions can be handled.

`noexcept` is most useful (and most clearly correct) for frequently used, low-level functions.

Note Destructors, `swap` functions, move operations, and default constructors should never throw.

Enforcement

- Flag functions that are not `noexcept`, yet cannot throw.
- Flag throwing `swap`, `move`, destructors, and default constructors.

F.7: For general use, take `T*` or `T&` arguments rather than smart pointers

Reason Passing a smart pointer transfers or shares ownership and should only be used when ownership semantics are intended (see R.30). Passing by smart pointer restricts the use of a function to callers that use smart pointers. Passing a shared smart pointer (e.g., `std::shared_ptr`) implies a run-time cost.

Example

```
void f(int*);              // accepts any int*
void g(unique_ptr<int>);   // can only accept ints for which you want to transfer owners
void g(shared_ptr<int>);   // can only accept ints for which you are willing to share ou

void h(const unique_ptr<int>&);  // doesn't change ownership, but requires a particular

void h(int&);              // accepts any int
```

Example, bad

```
// callee
void f(shared_ptr<widget>& w)
{
    // ...
    use(*w); // only use of w -- the lifetime is not used at all
    // ...
};
```

See further in R.30.

Note We can catch dangling pointers statically, so we don't need to rely on resource management to avoid violations from dangling pointers.

See also: when to prefer `T*` and when to prefer `T&`.

See also: Discussion of smart pointer use.

Enforcement

- Flag a parameter of a smart pointer type (a type that overloads `operator->` or `operator*`) that is copyable but never copied/moved from in the function body or else movable but never moved from in the function body or by being a by-value parameter, and that is never modified, and that is not passed along to another function that could do so. That means the ownership semantics are not used.

F.8: Prefer pure functions

Reason Pure functions are easier to reason about, sometimes easier to optimize (and even parallelize), and sometimes can be memoized.

Example

```
template<class T>
auto square(T t) { return t * t; }
```

Note `constexpr` functions are pure.

Enforcement Not possible.

F.call: Parameter passing

There are a variety of ways to pass parameters to a function and to return values.

F.15: Prefer simple and conventional ways of passing information

Reason Using "unusual and clever" techniques causes surprises, slows understanding by other programmers, and encourages bugs. If you really feel the need for an optimization beyond the common techniques, measure to ensure that it really is an improvement, and document/comment because the improvement may not be portable.

The following tables summarize the advice in the following Guidelines, F.16-21.

<table>
<tr><th></th><th>Cheap or impossible to copy (e.g., int, unique_ptr)</th><th>Cheap to move (e.g., vector<T>, string)
or Moderate cost to move (e.g., array<vector>, BigPOD)
or Don't know (e.g., unfamiliar type, template)</th><th>Expensive to move (e.g., BigPOD[], array<BigPOD>)</th></tr>
<tr><td>Out</td><td colspan="2">X f()</td><td></td></tr>
<tr><td>In/Out</td><td colspan="3">f(X&)</td></tr>
<tr><td>In</td><td rowspan="2">f(X)</td><td colspan="2" rowspan="2">f(const X&)</td></tr>
<tr><td>In & retain "copy"</td></tr>
</table>

"Cheap" ≈ a handful of hot int copies
"Moderate cost" ≈ memcpy hot/contiguous ~1KB and no allocation

** or return unique_ptr<X>/make_shared_<X> at the cost of a dynamic allocation*

Figure 1: Normal parameter passing table

<table>
<tr><th></th><th>Cheap or impossible to copy (e.g., int, unique_ptr)</th><th>Cheap to move (e.g., vector<T>, string)
or Moderate cost to move (e.g., array<vector>, BigPOD)
or Don't know (e.g., unfamiliar type, template)</th><th>Expensive to move (e.g., BigPOD[], array<BigPOD>)</th></tr>
<tr><td>Out</td><td colspan="2">X f()</td><td></td></tr>
<tr><td>In/Out</td><td colspan="3">f(X&)</td></tr>
<tr><td>In</td><td rowspan="2">f(X)</td><td colspan="2">f(const X&)</td></tr>
<tr><td>In & retain copy</td><td>f(const X&) + f(X&&) & move **</td><td></td></tr>
<tr><td>In & move from</td><td colspan="3">f(X&&) **</td></tr>
</table>

** or return unique_ptr<X>/make_shared_<X> at the cost of a dynamic allocation*

*** special cases can also use perfect forwarding (e.g., multiple in+copy params, conversions)*

Figure 2: Advanced parameter passing table

F.16: For "in" parameters, pass cheaply-copied types by value and others by reference to `const`

Reason Both let the caller know that a function will not modify the argument, and both allow initialization by rvalues.

What is "cheap to copy" depends on the machine architecture, but two or three words (doubles, pointers, references) are usually best passed by value. When copying is cheap, nothing beats the simplicity and safety of copying, and for small objects (up to two or three words) it is also faster than passing by reference because it does not require an extra reference to access from the function.

Example

```
void fct(const string& s);  // OK: pass by reference to const; always cheap

void fct2(string s);        // bad: potentially expensive

void fct(int x);          // OK: Unbeatable

void fct2(const int& x);  // bad: overhead on access in fct2()
```

For advanced uses (only), where you really need to optimize for rvalues passed to "input-only" parameters:

- If the function is going to unconditionally move from the argument, take it by `&&`. See F.18.
- If the function is going to keep a copy of the argument, in addition to passing by `const&` (for lvalues), add an overload that passes the parameter by `&&` (for rvalues) and in the body `std::move`s it to its destination. Essentially this overloads a "consume"; see F.18.
- In special cases, such as multiple "input + copy" parameters, consider using perfect forwarding. See F.19.

Example

```
int multiply(int, int); // just input ints, pass by value

string& concatenate(string&, const string& suffix); // suffix is input-only but not as

void sink(unique_ptr<widget>);  // input only, and consumes the widget
```

Avoid "esoteric techniques" such as:

- Passing arguments as `T&&` "for efficiency". Most rumors about performance advantages from passing by `&&` are false or brittle (but see F.25.)
- Returning `const T&` from assignments and similar operations (see F.47.)

Example Assuming that `Matrix` has move operations (possibly by keeping its elements in a `std::vector`):

```
Matrix operator+(const Matrix& a, const Matrix& b)
{
    Matrix res;
    // ... fill res with the sum ...
    return res;
}

Matrix x = m1 + m2;  // move constructor

y = m3 + m3;         // move assignment
```

Notes The return value optimization doesn't handle the assignment case, but the move assignment does.

A reference may be assumed to refer to a valid object (language rule). There is no (legitimate) "null reference." If you need the notion of an optional value, use a pointer, `std::optional`, or a special value used to denote "no value."

Enforcement

- (Simple) ((Foundation)) Warn when a parameter being passed by value has a size greater than `4 * sizeof(int)`. Suggest using a reference to `const` instead.
- (Simple) ((Foundation)) Warn when a `const` parameter being passed by reference has a size less than `3 * sizeof(int)`. Suggest passing by value instead.
- (Simple) ((Foundation)) Warn when a `const` parameter being passed by reference is `move`d.

F.17: For "in-out" parameters, pass by reference to non-`const`

Reason This makes it clear to callers that the object is assumed to be modified.

Example

```
void update(Record& r);  // assume that update writes to r
```

Note A `T&` argument can pass information into a function as well as well as out of it. Thus `T&` could be an in-out-parameter. That can in itself be a problem and a source of errors:

```
void f(string& s)
{
    s = "New York";   // non-obvious error
}

void g()
{
    string buffer = ".................................";
    f(buffer);
    // ...
}
```

Here, the writer of `g()` is supplying a buffer for `f()` to fill, but `f()` simply replaces it (at a somewhat higher cost than a simple copy of the characters). If the writer of `g()` makes an assumption about the size of `buffer` a bad logic error can happen.

Enforcement

- (Moderate) ((Foundation)) Warn about functions with reference to non-`const` parameters that do *not* write to them.
- (Simple) ((Foundation)) Warn when a non-`const` parameter being passed by reference is `move`d.

F.18: For "consume" parameters, pass by `X&&` and `std::move` the parameter

Reason It's efficient and eliminates bugs at the call site: `X&&` binds to rvalues, which requires an explicit `std::move` at the call site if passing an lvalue.

Example

```
???
```

Exception Unique owner types that are move-only and cheap-to-move, such as `unique_ptr`, can also be passed by value which is simpler to write and achieves the same effect. Passing by value does generate one extra (cheap) move operation, but prefer simplicity and clarity first.

Enforcement

- Flag all `X&&` parameters (where `X` is not a template type parameter name) where the function body uses them without `std::move`.
- Flag access to moved-from objects.
- Don't conditionally move from objects

F.19: For "forward" parameters, pass by `TP&&` and only `std::forward` the parameter

Reason If the object is to be passed onward to other code and not directly used by this function, we want to make this function agnostic to the argument `const`-ness and rvalue-ness.

In that case, and only that case, make the parameter `TP&&` where `TP` is a template type parameter – it both *ignores* and *preserves* `const`-ness and rvalue-ness. Therefore any code that uses a `TP&&` is implicitly declaring that it itself doesn't care about the variable's `const`-ness and rvalue-ness (because it is ignored), but that intends to pass the value onward to other code that does care about `const`-ness and rvalue-ness (because it is preserved). When used as a parameter `TP&&` is safe because any temporary objects passed from the caller will live for the duration of the function call. A parameter of type `TP&&` should essentially always be passed onward via `std::forward` in the body of the function.

Example

```
template <class F, class... Args>
inline auto invoke(F f, Args&&... args) {
    return f(forward<Args>(args)...);
}

??? calls ???
```

Enforcement

- Flag a function that takes a `TP&&` parameter (where `TP` is a template type parameter name) and does anything with it other than `std::forward`ing it exactly once on every static path.

F.20: For "out" output values, prefer return values to output parameters

Reason A return value is self-documenting, whereas a `&` could be either in-out or out-only and is liable to be misused.

This includes large objects like standard containers that use implicit move operations for performance and to avoid explicit memory management.

If you have multiple values to return, use a tuple or similar multi-member type.

Example

```
vector<const int*> find_all(const vector<int>&, int x);  // OK: return pointers to elem

void find_all(const vector<int>&, vector<const int*>& out, int x);  // Bad: place point
```

Note A struct of many (individually cheap-to-move) elements may be in aggregate expensive to move.

It is not recommended to return a `const` value. Such older advice is now obsolete; it does not add value, and it interferes with move semantics.

```
??? example ???
```

Exceptions

- For non-value types, such as types in an inheritance hierarchy, return the object by `unique_ptr` or `shared_ptr`.
- If a type is expensive to move (e.g., `array<BigPOD>`), consider allocating it on the free store and return a handle (e.g., `unique_ptr`), or passing it in a reference to non-`const` target object to fill (to be used as an out-parameter).
- In the special case of allowing a caller to reuse an object that carries capacity (e.g., `std::string`, `std::vector`) across multiple calls to the function in an inner loop, treat it as an in/out parameter instead and pass by `&`. This is one use of the more generally named "caller-allocated out" pattern.

Example

```
struct Package {      // exceptional case: expensive-to-move object
    char header[16];
    char load[2024 - 16];
};

Package fill();       // Bad: large return value
void fill(Package&);  // OK

int val();            // OK
void val(int&);       // Bad: Is val reading its argument
```

Enforcement

- Flag reference to non-`const` parameters that are not read before being written to and are a type that could be cheaply returned; they should be "out" return values.
- Flag returning a `const` value. To fix: Remove `const` to return a non-`const` value instead.

F.21: To return multiple "out" values, prefer returning a tuple or struct

Reason A return value is self-documenting as an "output-only" value. And yes, C++ does have multiple return values, by convention of using a `tuple`, with the extra convenience of `tie` at the call site.

Example

```
int f(const string& input, /*output only*/ string& output_data) // BAD: output-only par
{
    // ...
    output_data = something();
    return status;
}

tuple<int, string> f(const string& input) // GOOD: self-documenting
{
    // ...
    return make_tuple(status, something());
}
```

In fact, C++98's standard library already used this convenient feature, because a `pair` is like a two-element `tuple`. For example, given a `set<string> myset`, consider:

```
// C++98
result = myset.insert("Hello");
if (result.second) do_something_with(result.first);    // workaround
```

With C++11 we can write this, putting the results directly in existing local variables:

```
Sometype iter;                                // default initialize if we hav
Someothertype success;                        // used these variables for som

tie(iter, success) = myset.insert("Hello");   // normal return value
if (success) do_something_with(iter);
```

With C++17 we may be able to write something like this, also declaring the variables:

```
auto { iter, success } = myset.insert("Hello");
if (success) do_something_with(iter);
```

Exception: For types like `string` and `vector` that carry additional capacity, it can sometimes be useful to treat it as in/out instead by using the "caller-allocated out" pattern, which is to pass an output-only object by reference to non-`const` so that when the callee writes to it the object can reuse any capacity or other resources that it already contains. This technique can dramatically reduce the number of allocations in a loop that repeatedly calls other functions to get string values, by using a single string object for the entire loop.

```
??? example ???
```

Note In some cases it may be useful to return a specific, user-defined `Value_or_error` type along the lines of `variant<T, error_code>`, rather than using the generic `tuple`.

Enforcement

- Output parameters should be replaced by return values. An output parameter is one that the function writes to, invokes a non-`const` member function, or passes on as a non-`const`.

F.22: Use `T*` or `owner<T*>` to designate a single object

Reason In traditional C and C++ code, plain `T*` is used for many weakly-related purposes, such as:

- Identify a (single) object (not to be deleted by this function)
- Point to an object allocated on the free store (and delete it later)
- Hold the `nullptr`
- Identify a C-style string (zero-terminated array of characters)
- Identify an array with a length specified separately
- Identify a location in an array

Example

```
void use(int* p, char* s, int* q)
{
        *++p = 666;   // Bad: we don't know if p points to two elements; assume it does
        cout << s;    // Bad: we don't know if that s points to a zero-terminated array
        delete q;     // Bad: we don't know if *q is allocated on the free store; assum
}
```

Note `owner<T*>` represents ownership, `zstring` represents a C-style string.

Also: Assume that a `T*` obtained from a smart pointer to `T` (e.g., `unique_ptr<T>`) points to a single element.

See also: Support library.

Enforcement

- (Simple) ((Bounds)) Warn for any arithmetic operation on an expression of pointer type that results in a value of pointer type.

F.23: Use a `not_null<T>` to indicate that "null" is not a valid value

Reason Clarity. A function with a `not_null<T>` parameter makes it clear that the caller of the function is responsible for any `nullptr` checks that may be necessary. Similarly, a function with a return value of `not_null<T>` makes it clear that the caller of the function does not need to check for `nullptr`.

Example `not_null<T*>` makes it obvious to a reader (human or machine) that a test for `nullptr` is not necessary before dereference. Additionally, when debugging, `owner<T*>` and `not_null<T>` can be instrumented to check for correctness.

Consider:

```
int length(Record* p);
```

When I call `length(p)` should I test for `p == nullptr` first? Should the implementation of `length()` test for `p == nullptr`?

```
int length(not_null<Record*> p);  // it is the caller's job to make sure p != nullptr

int length(Record* p);            // the implementor of length() must assume that p ==
```

Note A `not_null<T*>` is assumed not to be the `nullptr`; a `T*` may be the `nullptr`; both can be represented in memory as a `T*` (so no run-time overhead is implied).

Note `not_null` is not just for built-in pointers. It works for `unique_ptr`, `shared_ptr`, and other pointer-like types.

Enforcement

- (Simple) Warn if a raw pointer is dereferenced without being tested against `nullptr` (or equivalent) within a function, suggest it is declared `not_null` instead.
- (Simple) Error if a raw pointer is sometimes dereferenced after first being tested against `nullptr` (or equivalent) within the function and sometimes is not.
- (Simple) Warn if a `not_null` pointer is tested against `nullptr` within a function.

F.24: Use a `span<T>` or a `span_p<T>` to designate a half-open sequence

Reason Informal/non-explicit ranges are a source of errors.

Example

```
X* find(span<X> r, const X& v);    // find v in r

vector<X> vec;
// ...
auto p = find({vec.begin(), vec.end()}, X{});  // find X{} in vec
```

Note Ranges are extremely common in C++ code. Typically, they are implicit and their correct use is very hard to ensure. In particular, given a pair of arguments `(p, n)` designating an array [`p`:`p+n`), it is in general impossible to know if there really are `n` elements to access following `*p`. `span<T>` and `span_p<T>` are simple helper classes designating a [`p`:`q`) range and a range starting with `p` and ending with the first element for which a predicate is true, respectively.

Example A `span` represents a range of elements, but how do we manipulate elements of that range?

```
void f(span<int> s)
{
    for (int x : s) cout << x << '\n';  // range traversal (guaranteed correct)
    for (int i = 0; i<s.size(); ++i) cout << x << '\n';  // C-style traversal (potentia
    s[7] = 9;                           // random access (potentially checked)
    std::sort(&s[0],&s[s.size()/2]);    // extract pointers (potentially checked)
}
```

Note A `span<T>` object does not own its elements and is so small that it can be passed by value.

Passing a `span` object as an argument is exactly as efficient as passing a pair of pointer arguments or passing a pointer and an integer count.

See also: Support library.

Enforcement (Complex) Warn where accesses to pointer parameters are bounded by other parameters that are integral types and suggest they could use `span` instead.

F.25: Use a `zstring` or a `not_null<zstring>` to designate a C-style string

Reason C-style strings are ubiquitous. They are defined by convention: zero-terminated arrays of characters. We must distinguish C-style strings from a pointer to a single character or an old-fashioned pointer to an array of characters.

Example Consider:

```
int length(const char* p);
```

When I call `length(s)` should I test for `s == nullptr` first? Should the implementation of `length()` test for `p == nullptr`?

```
int length(zstring p);            // the implementor of length() must assume that p ==

int length(not_null<zstring> p);  // it is the caller's job to make sure p != nullptr
```

Note `zstring` do not represent ownership.

See also: Support library.

F.26: Use a `unique_ptr<T>` to transfer ownership where a pointer is needed

Reason Using `unique_ptr` is the cheapest way to pass a pointer safely.

Example

```
unique_ptr<Shape> get_shape(istream& is)  // assemble shape from input stream
{
    auto kind = read_header(is); // read header and identify the next shape on input
    switch (kind) {
    case kCircle:
        return make_unique<Circle>(is);
    case kTriangle:
        return make_unique<Triangle>(is);
    // ...
    }
}
```

Note You need to pass a pointer rather than an object if what you are transferring is an object from a class hierarchy that is to be used through an interface (base class).

Enforcement (Simple) Warn if a function returns a locally-allocated raw pointer. Suggest using either `unique_ptr` or `shared_ptr` instead.

F.27: Use a `shared_ptr<T>` to share ownership

Reason Using `std::shared_ptr` is the standard way to represent shared ownership. That is, the last owner deletes the object.

Example

```
shared_ptr<const Image> im { read_image(somewhere) };

std::thread t0 {shade, args0, top_left, im};
std::thread t1 {shade, args1, top_right, im};
std::thread t2 {shade, args2, bottom_left, im};
std::thread t3 {shade, args3, bottom_right, im};

// detach threads
// last thread to finish deletes the image
```

Note Prefer a `unique_ptr` over a `shared_ptr` if there is never more than one owner at a time. `shared_ptr` is for shared ownership.

Note that pervasive use of `shared_ptr` has a cost (atomic operations on the `shared_ptr`'s reference count have a measurable aggregate cost).

Alternative Have a single object own the shared object (e.g. a scoped object) and destroy that (preferably implicitly) when all users have completed.

Enforcement (Not enforceable) This is a too complex pattern to reliably detect.

F.60: Prefer `T*` over `T&` when "no argument" is a valid option

Reason A pointer (`T*`) can be a `nullptr` and a reference (`T&`) cannot, there is no valid "null reference". Sometimes having `nullptr` as an alternative to indicated "no object" is useful, but if it is not, a reference is notationally simpler and might yield better code.

Example

```
string zstring_to_string(zstring p) // zstring is a char*; that is a C-style string
{
    if (p==nullptr) return string{};    // p might be nullptr; remember to check
    return string{p};
}

void print(const vector<int>& r)
{
    // r refers to a vector<int>; no check needed
}
```

Note It is possible, but not valid C++ to construct a reference that is essentially a `nullptr` (e.g., `T* p = nullptr; T& r = (T&)*p;`). That error is very uncommon.

Note If you prefer the pointer notation (`->` and/or `*` vs. `.`), `not_null<T*>` provides the same guarantee as `T&`.

Enforcement

- Flag ???

F.42: Return a `T*` to indicate a position (only)

Reason That's what pointers are good for. Returning a `T*` to transfer ownership is a misuse.

Example

```
Node* find(Node* t, const string& s)  // find s in a binary tree of Nodes
{
    if (t == nullptr || t->name == s) return t;
    if ((auto p = find(t->left, s))) return p;
    if ((auto p = find(t->right, s))) return p;
    return nullptr;
}
```

If it isn't the `nullptr`, the pointer returned by `find` indicates a `Node` holding `s`. Importantly, that does not imply a transfer of ownership of the pointed-to object to the caller.

Note Positions can also be transferred by iterators, indices, and references. A reference is often a superior alternative to a pointer if there is no need to use `nullptr` or if the object referred to should not change.

Note Do not return a pointer to something that is not in the caller's scope; see F.43.

Example, bad

```
int* f()
{
    int x = 7;
    // ...
    return &x;  // Bad: returns pointer to object that is about to be destroyed
}
```

This applies to references as well:

```
int& f()
{
    int x = 7;
    // ...
    return x;  // Bad: returns reference to object that is about to be destroyed
}
```

See also: discussion of dangling pointer prevention.

Enforcement A slightly different variant of the problem is placing pointers in a container that outlives the objects pointed to.

- Compilers tend to catch return of reference to locals and could in many cases catch return of pointers to locals.
- Static analysis can catch many common patterns of the use of pointers indicating positions (thus eliminating dangling pointers)

F.43: Never (directly or indirectly) return a pointer to a local object

Reason To avoid the crashes and data corruption that can result from the use of such a dangling pointer.

Example, bad After the return from a function its local objects no longer exist:

```
int* f()
{
    int fx = 9;
    return &fx;  // BAD
}

void g(int* p)   // looks innocent enough
{
    int gx;
    cout << "*p == " << *p << '\n';
    *p = 999;
    cout << "gx == " << gx << '\n';
}

void h()
{
    int* p = f();
    int z = *p;     // read from abandoned stack frame (bad)
    g(p);           // pass pointer to abandoned stack frame to function (bad)
}
```

Here on one popular implementation I got the output:

```
*p == 999
gx == 999
```

I expected that because the call of `g()` reuses the stack space abandoned by the call of `f()` so `*p` refers to the space now occupied by `gx`.

Imagine what would happen if `fx` and `gx` were of different types. Imagine what would happen if `fx` or `gx` was a type with an invariant. Imagine what would happen if more that dangling pointer was passed around among a larger set of functions. Imagine what a cracker could do with that dangling pointer.

Fortunately, most (all?) modern compilers catch and warn against this simple case.

Note You can construct similar examples using references.

Note This applies only to non-`static` local variables. All `static` variables are (as their name indicates) statically allocated, so that pointers to them cannot dangle.

Example, bad Not all examples of leaking a pointer to a local variable are that obvious:

```
int* glob;       // global variables are bad in so many ways

template<class T>
void steal(T x)
{
    glob = x();  // BAD
}

void f()
{
    int i = 99;
    steal([&] { return &i; });
}

int main()
{
    f();
    cout << *glob << '\n';
}
```

Here I managed to read the location abandoned by the call of `f`. The pointer stored in `glob` could be used much later and cause trouble in unpredictable ways.

Note The address of a local variable can be "returned"/leaked by a return statement, by a `T&` out-parameter, as a member of a returned object, as an element of a returned array, and more.

Note Similar examples can be constructed "leaking" a pointer from an inner scope to an outer one; such examples are handled equivalently to leaks of pointers out of a function.

See also: Another way of getting dangling pointers is pointer invalidation. It can be detected/prevented with similar techniques.

Enforcement Preventable through static analysis.

F.44: Return a `T&` when copy is undesirable and "returning no object" isn't needed

Reason The language guarantees that a `T&` refers to an object, so that testing for `nullptr` isn't necessary.

See also: The return of a reference must not imply transfer of ownership: discussion of dangling pointer prevention and discussion of ownership.

Example

```
class car
{
    array<wheel,4> w;
    // ...
public:
    wheel& get_wheel(size_t i) { Expects(i<4); return w[i]; }
    // ...
};

void use()
{
    car c;
    wheel& w0 = c.get_wheel(0); // w0 has the same lifetime as c
}
```

Enforcement Flag functions where no `return` expression could yield `nullptr`

F.45: Don't return a `T&&`

Reason It's asking to return a reference to a destroyed temporary object. A `&&` is a magnet for temporary objects. This is fine when the reference to the temporary is being passed "downward" to a callee, because the temporary is guaranteed to outlive the function call. (See F.24 and F.25.) However, it's not fine when passing such a reference "upward" to a larger caller scope. See also ???.

For passthrough functions that pass in parameters (by ordinary reference or by perfect forwarding) and want to return values, use simple `auto` return type deduction (not `auto&&`).

Example, bad If `F` returns by value, this function returns a reference to a temporary.

```
template<class F>
auto&& wrapper(F f)
{
    log_call(typeid(f)); // or whatever instrumentation
    return f();
}
```

Example, good Better:

```
template<class F>
auto wrapper(F f)
```

```
{
    log_call(typeid(f)); // or whatever instrumentation
    return f();
}
```

Exception: `std::move` and `std::forward` do return `&&`, but they are just casts – used by convention only in expression contexts where a reference to a temporary object is passed along within the same expression before the temporary is destroyed. We don't know of any other good examples of returning `&&`.

Enforcement Flag any use of `&&` as a return type, except in `std::move` and `std::forward`.

F.46: `int` is the return type for `main()`

Reason It's a language rule, but violated through "language extensions" so often that it is worth mentioning. Declaring `main` (the one global `main` of a program) `void` limits portability.

Example

```
void main() { /* ... */ };  // bad, not C++

int main()
{
    std::cout << "This is the way to do it\n";
}
```

Note We mention this only because of the persistence of this error in the community.

Enforcement

- The compiler should do it
- If the compiler doesn't do it, let tools flag it

F.47: Return `T&` from assignment operators

Reason The convention for operator overloads (especially on value types) is for `operator=(const T&)` to perform the assignment and then return (non-const) `*this`. This ensures consistency with standard library types and follows the principle of "do as the ints do."

Note Historically there was some guidance to make the assignment operator return `const T&`. This was primarily to avoid code of the form `(a=b)=c` - such code is not common enough to warrant violating consistency with standard types.

Example

```
class Foo
{
 public:
    ...
    Foo& operator=(const Foo& rhs) {
      // Copy members.
      ...
      return *this;
    }
};
```

Enforcement This should be enforced by tooling by checking the return type (and return value) of any assignment operator.

F.50: Use a lambda when a function won't do (to capture local variables, or to write a local function)

Reason Functions can't capture local variables or be declared at local scope; if you need those things, prefer a lambda where possible, and a handwritten function object where not. On the other hand, lambdas and function objects don't overload; if you need to overload, prefer a function (the workarounds to make lambdas overload are ornate). If either will work, prefer writing a function; use the simplest tool necessary.

Example

```
// writing a function that should only take an int or a string -- overloading is natur
void f(int);
void f(const string&);

// writing a function object that needs to capture local state and appear
// at statement or expression scope -- a lambda is natural
vector<work> v = lots_of_work();
for (int tasknum = 0; tasknum < max; ++tasknum) {
    pool.run([=, &v]{
        /*
        ...
```

```
        ... process 1/max-th of v, the tasknum-th chunk
        ...
        */
    });
}
pool.join();
```

Exception: Generic lambdas offer a concise way to write function templates and so can be useful even when a normal function template would do equally well with a little more syntax. This advantage will probably disappear in the future once all functions gain the ability to have Concept parameters.

Enforcement

- Warn on use of a named non-generic lambda (e.g., `auto x = [](int i){ /*...*/; };`) that captures nothing and appears at global scope. Write an ordinary function instead.

F.51: Where there is a choice, prefer default arguments over overloading

Reason Default arguments simply provides alternative interfaces to a single implementation. There is no guarantee that a set of overloaded functions all implement the same semantics. The use of default arguments can avoid code replication.

Note There is a choice between using default argument and overloading when the alternatives are from a set of arguments of the same types. For example:

```
void print(const string& s, format f = {});
```

as opposed to

```
void print(const string& s);  // use default format
void print(const string& s, format f);
```

There is not a choice when a set of functions are used to do a semantically equivalent operation to a set of types. For example:

```
void print(const char&);
void print(int);
void print(zstring);
```

See also

```
[Default arguments for virtual functions](#Rf-virtual-default-arg}
```

Enforcement

```
???
```

F.52: Prefer capturing by reference in lambdas that will be used locally, including passed to algorithms

Reason For efficiency and correctness, you nearly always want to capture by reference when using the lambda locally. This includes when writing or calling parallel algorithms that are local because they join before returning.

Example This is a simple three-stage parallel pipeline. Each `stage` object encapsulates a worker thread and a queue, has a `process` function to enqueue work, and in its destructor automatically blocks waiting for the queue to empty before ending the thread.

```
void send_packets(buffers& bufs)
{
    stage encryptor  ([] (buffer& b){ encrypt(b); });
    stage compressor ([&](buffer& b){ compress(b); encryptor.process(b); });
    stage decorator  ([&](buffer& b){ decorate(b); compressor.process(b); });
    for (auto& b : bufs) { decorator.process(b); }
}  // automatically blocks waiting for pipeline to finish
```

Enforcement ???

F.53: Avoid capturing by reference in lambdas that will be used nonlocally, including returned, stored on the heap, or passed to another thread

Reason Pointers and references to locals shouldn't outlive their scope. Lambdas that capture by reference are just another place to store a reference to a local object, and shouldn't do so if they (or a copy) outlive the scope.

Example

```
{
    // ...

    // a, b, c are local variables
    background_thread.queue_work([=]{ process(a, b, c); });  // want copies of a, b, an
}
```

Enforcement ???

F.54: If you capture `this`, capture all variables explicitly (no default capture)

Reason It's confusing. Writing `[=]` in a member function appears to capture by value, but actually captures data members by reference because it actually captures the invisible `this` pointer by value. If you meant to do that, write `this` explicitly.

Example

```
class myclass {
    int x = 0;
    // ...

    void f() {
        int i = 0;
        // ...

        auto lambda = [=]{ use(i,x); };   // BAD: "looks like" copy/value capture
            // notes: [&] has identical semantics and copies the this pointer under th
            //        [=,this] and [&,this] are not much better, and confusing
        x = 42;
        lambda(); // calls use(42);
        x = 43;
        lambda(); // calls use(43);

        // ...

        auto lambda2 = [i,this]{ use(i,x); }; // ok, most explicit and least confusing

        // ...
    }
};
```

Note This is under active discussion in standardization, and may be addressed in a future version of the standard by adding a new capture mode or possibly adjusting the meaning of `[=]`. For now, just be explicit.

Enforcement

- Flag any lambda capture-list that specifies a default capture and also captures `this` (whether explicitly or via default capture)

C: Classes and Class Hierarchies

A class is a user-defined type, for which a programmer can define the representation, operations, and interfaces. Class hierarchies are used to organize related classes into hierarchical structures.

Class rule summary:

- C.1: Organize related data into structures (`struct`s or `class`es)
- C.2: Use `class` if the class has an invariant; use `struct` if the data members can vary independently
- C.3: Represent the distinction between an interface and an implementation using a class
- C.4: Make a function a member only if it needs direct access to the representation of a class
- C.5: Place helper functions in the same namespace as the class they support
- C.7: Don't define a class or enum and declare a variable of its type in the same statement
- C.8: use `class` rather that `struct` if any member is non-public
- C.9: minimize exposure of members

Subsections:

- C.concrete: Concrete types
- C.ctor: Constructors, assignments, and destructors
- C.con: Containers and other resource handles
- C.lambdas: Function objects and lambdas
- C.hier: Class hierarchies (OOP)
- C.over: Overloading and overloaded operators
- C.union: Unions

C.1: Organize related data into structures (`struct`s or `class`es)

Reason Ease of comprehension. If data is related (for fundamental reasons), that fact should be reflected in code.

Example

```
void draw(int x, int y, int x2, int y2);  // BAD: unnecessary implicit relationships
void draw(Point from, Point to);          // better
```

Note A simple class without virtual functions implies no space or time overhead.

Note From a language perspective `class` and `struct` differ only in the default visibility of their members.

Enforcement Probably impossible. Maybe a heuristic looking for data items used together is possible.

C.2: Use `class` if the class has an invariant; use `struct` if the data members can vary independently

Reason Readability. Ease of comprehension. The use of `class` alerts the programmer to the need for an invariant. This is a useful convention.

Note An invariant is a logical condition for the members of an object that a constructor must establish for the public member functions to assume. After the invariant is established (typically by a constructor) every member function can be called for the object. An invariant can be stated informally (e.g., in a comment) or more formally using `Expects`.

If all data members can vary independently of each other, no invariant is possible.

Example

```
struct Pair {  // the members can vary independently
    string name;
    int volume;
};
```

but:

```
class Date {
public:
    Date(int yy, Month mm, char dd);    // validate that {yy, mm, dd} is a valid date a
    // ...
private:
    int y;
    Month m;
    char d;    // day
};
```

Note If a class has any `private` data, a user cannot completely initialize an object without the use of a constructor. Hence, the class definer will provide a constructor and must specify its meaning. This effectively means the definer need to define an invariant.

- See also define a class with private data as `class`.
- See also Prefer to place the interface first in a class.
- See also minimize exposure of members.
- See also Avoid `protected` data.

Enforcement Look for `struct`s with all data private and `class`es with public members.

C.3: Represent the distinction between an interface and an implementation using a class

Reason An explicit distinction between interface and implementation improves readability and simplifies maintenance.

Example

```
class Date {
    // ... some representation ...
public:
    Date();
    Date(int yy, Month mm, char dd);    // validate that {yy, mm, dd} is a valid date a

    int day() const;
    Month month() const;
    // ...
};
```

For example, we can now change the representation of a `Date` without affecting its users (recompilation is likely, though).

Note Using a class in this way to represent the distinction between interface and implementation is of course not the only way. For example, we can use a set of declarations of freestanding functions in a namespace, an abstract base class, or a template function with concepts to represent an interface. The most important issue is to explicitly distinguish between an interface and its implementation "details." Ideally, and typically, an interface is far more stable than its implementation(s).

Enforcement ???

C.4: Make a function a member only if it needs direct access to the representation of a class

Reason Less coupling than with member functions, fewer functions that can cause trouble by modifying object state, reduces the number of functions that needs to be modified after a change in representation.

Example

```
class Date {
    // ... relatively small interface ...
};

// helper functions:
Date next_weekday(Date);
bool operator==(Date, Date);
```

The "helper functions" have no need for direct access to the representation of a `Date`.

Note This rule becomes even better if C++17 gets "uniform function call." ???

Enforcement Look for member function that do not touch data members directly. The snag is that many member functions that do not need to touch data members directly do.

C.5: Place helper functions in the same namespace as the class they support

Reason A helper function is a function (usually supplied by the writer of a class) that does not need direct access to the representation of the class, yet is seen as part of the useful interface to the class. Placing them in the same namespace as the class makes their relationship to the class obvious and allows them to be found by argument dependent lookup.

Example

```
namespace Chrono { // here we keep time-related services

    class Time { /* ... */ };
    class Date { /* ... */ };

    // helper functions:
    bool operator==(Date, Date);
    Date next_weekday(Date);
    // ...
}
```

Note This is expecially important for overloaded operators.

Enforcement

- Flag global functions taking argument types from a single namespace.

C.7: Don't define a class or enum and declare a variable of its type in the same statement

Reason Mixing a type definition and the definition of another entity in the same declaration is confusing and unnecessary.

Example; bad

```
struct Data { /*...*/ } data{ /*...*/ };
```

Example; good

```
struct Data { /*...*/ };
Data data{ /*...*/ };
```

Enforcement

- Flag if the `}` of a class or enumeration definition is not followed by a `;`. The `;` is missing.

C.8: use `class` rather that `struct` if any member is non-public

Reason Readability. To make it clear that something is being hidden/abstracted. This is a useful convention.

Example, bad

```
struct Date {
    int d,m;

    Date(int i, Month m);
    // ... lots of functions ...
private:
    int y;  // year
};
```

There is nothing wrong with this code as far as the C++ language rules are concerned, but nearly everything is wrong from a design perspective. The private data is hidden far from the public data. The data is split in different parts of the class declaration. Different parts of the data has difference access. All of this decreases readability and complicates maintenance.

Note Prefer to place the interface first in a class see.

Enforcement Flag classes declared with `struct` if there is a `private` or `public` member.

C.9: minimize exposure of members

Reason Encapsulation. Information hiding. Minimize the chance of untended access. This simplifies maintenance.

Example

```
???
```

Note Prefer the order `public` members before `protected` members before `private` members see.

Enforcement ???

C.concrete: Concrete types

One ideal for a class is to be a regular type. That means roughly "behaves like an `int`." A concrete type is the simplest kind of class. A value of regular type can be copied and the result of a copy is an independent object with the same value as the original. If a concrete type has both `=` and `==`, `a=b` should result in `a == b` being `true`. Concrete classes without assignment and equality can be defined, but they are (and should be) rare. The C++ built-in types are regular, and so are standard-library classes, such as `string`, `vector`, and `map`. Concrete types are also often referred to as value types to distinguish them from types uses as part of a hierarchy.

Concrete type rule summary:

- C.10: Prefer a concrete type over more complicated classes
- C.11: Make concrete types regular

C.10 Prefer a concrete type over more complicated classes

Reason A concrete type is fundamentally simpler than a hierarchy: easier to design, easier to implement, easier to use, easier to reason about, smaller, and faster. You need a reason (use cases) for using a hierarchy.

Example

```
class Point1 {
    int x, y;
    // ... operations ...
    // ... no virtual functions ...
};

class Point2 {
    int x, y;
    // ... operations, some virtual ...
    virtual ~Point2();
};

void use()
{
    Point1 p11 {1, 2};   // make an object on the stack
    Point1 p12 {p11};    // a copy

    auto p21 = make_unique<Point2>(1, 2);   // make an object on the free store
    auto p22 = p21.clone();                 // make a copy
    // ...
}
```

If a class can be part of a hierarchy, we (in real code if not necessarily in small examples) must manipulate its objects through pointers or references. That implies more memory overhead, more allocations and deallocations, and more run-time overhead to perform the resulting indirections.

Note Concrete types can be stack allocated and be members of other classes.

Note The use of indirection is fundamental for run-time polymorphic interfaces. The allocation/deallocation overhead is not (that's just the most common case). We can use a base class as the interface of a scoped object of a derived class. This is done where dynamic allocation is prohibited (e.g. hard real-time) and to provide a stable interface to some kinds of plug-ins.

Enforcement ???

C.11: Make concrete types regular

Reason Regular types are easier to understand and reason about than types that are not regular (irregularities requires extra effort to understand and use).

Example

```
struct Bundle {
    string name;
    vector<Record> vr;
};

bool operator==(const Bundle& a, const Bundle& b) { return a.name == b.name && a.vr == b

Bundle b1 { "my bundle", {r1, r2, r3}};
Bundle b2 = b1;
if (!(b1 == b2)) error("impossible!");
b2.name = "the other bundle";
if (b1 == b2) error("No!");
```

In particular, if a concrete type has an assignment also give it an equals operator so that `a=b` implies `a == b`.

Enforcement ???

C.ctor: Constructors, assignments, and destructors

These functions control the lifecycle of objects: creation, copy, move, and destruction. Define constructors to guarantee and simplify initialization of classes.

These are *default operations*:

- a default constructor: `X()`
- a copy constructor: `X(const X&)`
- a copy assignment: `operator=(const X&)`
- a move constructor: `X(X&&)`
- a move assignment: `operator=(X&&)`
- a destructor: `~X()`

By default, the compiler defines each of these operations if it is used, but the default can be suppressed.

The default operations are a set of related operations that together implement the lifecycle semantics of an object. By default, C++ treats classes as value-like types, but not all types are value-like.

Set of default operations rules:

- C.20: If you can avoid defining any default operations, do
- C.21: If you define or `=delete` any default operation, define or `=delete` them all
- C.22: Make default operations consistent

Destructor rules:

- C.30: Define a destructor if a class needs an explicit action at object destruction
- C.31: All resources acquired by a class must be released by the class's destructor
- C.32: If a class has a raw pointer (`T*`) or reference (`T&`), consider whether it might be owning
- C.33: If a class has an owning pointer member, define or `=delete` a destructor
- C.34: If a class has an owning reference member, define or `=delete` a destructor
- C.35: A base class with a virtual function needs a virtual destructor
- C.36: A destructor may not fail
- C.37: Make destructors `noexcept`

Constructor rules:

- C.40: Define a constructor if a class has an invariant
- C.41: A constructor should create a fully initialized object
- C.42: If a constructor cannot construct a valid object, throw an exception
- C.43: Ensure that a class has a default constructor
- C.44: Prefer default constructors to be simple and non-throwing
- C.45: Don't define a default constructor that only initializes data members; use member initializers instead
- C.46: By default, declare single-argument constructors `explicit`
- C.47: Define and initialize member variables in the order of member declaration
- C.48: Prefer in-class initializers to member initializers in constructors for constant initializers
- C.49: Prefer initialization to assignment in constructors
- C.50: Use a factory function if you need "virtual behavior" during initialization
- C.51: Use delegating constructors to represent common actions for all constructors of a class
- C.52: Use inheriting constructors to import constructors into a derived class that does not need further explicit initialization

Copy and move rules:

- C.60: Make copy assignment non-`virtual`, take the parameter by `const&`, and return by non-`const&`
- C.61: A copy operation should copy
- C.62: Make copy assignment safe for self-assignment
- C.63: Make move assignment non-`virtual`, take the parameter by `&&`, and return by non-`const&`
- C.64: A move operation should move and leave its source in a valid state
- C.65: Make move assignment safe for self-assignment
- C.66: Make move operations `noexcept`
- C.67: A base class should suppress copying, and provide a virtual `clone` instead if "copying" is desired

Other default operations rules:

- C.80: Use `=default` if you have to be explicit about using the default semantics
- C.81: Use `=delete` when you want to disable default behavior (without wanting an alternative)
- C.82: Don't call virtual functions in constructors and destructors
- C.83: For value-like types, consider providing a `noexcept` swap function
- C.84: A `swap` may not fail
- C.85: Make `swap` `noexcept`
- C.86: Make `==` symmetric with respect of operand types and `noexcept`
- C.87: Beware of `==` on base classes
- C.89: Make a `hash` `noexcept`

C.defop: Default Operations

By default, the language supply the default operations with their default semantics. However, a programmer can disable or replace these defaults.

C.20: If you can avoid defining default operations, do

Reason It's the simplest and gives the cleanest semantics.

Example

```
struct Named_map {
public:
    // ... no default operations declared ...
```

```
private:
    string name;
    map<int, int> rep;
};

Named_map nm;        // default construct
Named_map nm2 {nm};  // copy construct
```

Since `std::map` and `string` have all the special functions, no further work is needed.

Note This is known as "the rule of zero".

Enforcement (Not enforceable) While not enforceable, a good static analyzer can detect patterns that indicate a possible improvement to meet this rule. For example, a class with a (pointer, size) pair of member and a destructor that `delete`s the pointer could probably be converted to a `vector`.

C.21: If you define or =delete any default operation, define or =delete them all

Reason The semantics of the special functions are closely related, so if one needs to be non-default, the odds are that others need modification too.

Example, bad

```
struct M2 {   // bad: incomplete set of default operations
public:
    // ...
    // ... no copy or move operations ...
    ~M2() { delete[] rep; }
private:
    pair<int, int>* rep;  // zero-terminated set of pairs
};

void use()
{
    M2 x;
    M2 y;
    // ...
    x = y;   // the default assignment
    // ...
}
```

Given that "special attention" was needed for the destructor (here, to deallocate), the likelihood that copy and move assignment (both will implicitly destroy an object) are correct is low (here, we would get double deletion).

Note This is known as "the rule of five" or "the rule of six", depending on whether you count the default constructor.

Note If you want a default implementation of a default operation (while defining another), write `=default` to show you're doing so intentionally for that function. If you don't want a default operation, suppress it with `=delete`.

Note Compilers enforce much of this rule and ideally warn about any violation.

Note Relying on an implicitly generated copy operation in a class with a destructor is deprecated.

Enforcement (Simple) A class should have a declaration (even a `=delete` one) for either all or none of the special functions.

C.22: Make default operations consistent

Reason The default operations are conceptually a matched set. Their semantics are interrelated. Users will be surprised if copy/move construction and copy/move assignment do logically different things. Users will be surprised if constructors and destructors do not provide a consistent view of resource management. Users will be surprised if copy and move don't reflect the way constructors and destructors work.

Example, bad

```
class Silly {   // BAD: Inconsistent copy operations
    class Impl {
        // ...
    };
    shared_ptr<Impl> p;
public:
    Silly(const Silly& a) : p{a.p} { *p = *a.p; }   // deep copy
    Silly& operator=(const Silly& a) { p = a.p; }   // shallow copy
    // ...
};
```

These operations disagree about copy semantics. This will lead to confusion and bugs.

Enforcement

- (Complex) A copy/move constructor and the corresponding copy/move assignment operator should write to the same member variables at the same level of dereference.
- (Complex) Any member variables written in a copy/move constructor should also be initialized by all other constructors.
- (Complex) If a copy/move constructor performs a deep copy of a member variable, then the destructor should modify the member variable.
- (Complex) If a destructor is modifying a member variable, that member variable should be written in any copy/move constructors or assignment operators.

C.dtor: Destructors

"Does this class need a destructor?" is a surprisingly powerful design question. For most classes the answer is "no" either because the class holds no resources or because destruction is handled by the rule of zero; that is, its members can take care of themselves as concerns destruction. If the answer is "yes", much of the design of the class follows (see the rule of five).

C.30: Define a destructor if a class needs an explicit action at object destruction

Reason A destructor is implicitly invoked at the end of an object's lifetime. If the default destructor is sufficient, use it. Only define a non-default destructor if a class needs to execute code that is not already part of its members' destructors.

Example

```
template<typename A>
struct final_action {   // slightly simplified
    A act;
    final_action(A a) :act{a} {}
    ~final_action() { act(); }
};

template<typename A>
final_action<A> finally(A act)   // deduce action type
{
    return final_action<A>{act};
}

void test()
{
    auto act = finally([]{ cout << "Exit test\n"; });  // establish exit action
```

```
    // ...
    if (something) return;   // act done here
    // ...
} // act done here
```

The whole purpose of `final_action` is to get a piece of code (usually a lambda) executed upon destruction.

Note There are two general categories of classes that need a user-defined destructor:

- A class with a resource that is not already represented as a class with a destructor, e.g., a `vector` or a transaction class.
- A class that exists primarily to execute an action upon destruction, such as a tracer or `final_action`.

Example, bad

```
class Foo {   // bad; use the default destructor
public:
    // ...
    ~Foo() { s = ""; i = 0; vi.clear(); }  // clean up
private:
    string s;
    int i;
    vector<int> vi;
};
```

The default destructor does it better, more efficiently, and can't get it wrong.

Note If the default destructor is needed, but its generation has been suppressed (e.g., by defining a move constructor), use `=default`.

Enforcement Look for likely "implicit resources", such as pointers and references. Look for classes with destructors even though all their data members have destructors.

C.31: All resources acquired by a class must be released by the class's destructor

Reason Prevention of resource leaks, especially in error cases.

Note For resources represented as classes with a complete set of default operations, this happens automatically.

Example

```
class X {
    ifstream f;   // may own a file
    // ... no default operations defined or =deleted ...
};
```

X's `ifstream` implicitly closes any file it may have open upon destruction of its `X`.

Example, bad

```
class X2 {     // bad
    FILE* f;   // may own a file
    // ... no default operations defined or =deleted ...
};
```

`X2` may leak a file handle.

Note What about a sockets that won't close? A destructor, close, or cleanup operation should never fail. If it does nevertheless, we have a problem that has no really good solution. For starters, the writer of a destructor does not know why the destructor is called and cannot "refuse to act" by throwing an exception. See discussion. To make the problem worse, many "close/release" operations are not retryable. Many have tried to solve this problem, but no general solution is known. If at all possible, consider failure to close/cleanup a fundamental design error and terminate.

Note A class can hold pointers and references to objects that it does not own. Obviously, such objects should not be `delete`d by the class's destructor. For example:

```
Preprocessor pp { /* ... */ };
Parser p { pp, /* ... */ };
Type_checker tc { p, /* ... */ };
```

Here `p` refers to `pp` but does not own it.

Enforcement

- (Simple) If a class has pointer or reference member variables that are owners (e.g., deemed owners by using `gsl::owner`), then they should be referenced in its destructor.
- (Hard) Determine if pointer or reference member variables are owners when there is no explicit statement of ownership (e.g., look into the constructors).

C.32: If a class has a raw pointer (`T*`) or reference (`T&`), consider whether it might be owning

Reason There is a lot of code that is non-specific about ownership.

Example

```
???
```

Note If the `T*` or `T&` is owning, mark it `owning`. If the `T*` is not owning, consider marking it `ptr`. This will aide documentation and analysis.

Enforcement Look at the initialization of raw member pointers and member references and see if an allocation is used.

C.33: If a class has an owning pointer member, define a destructor

Reason An owned object must be `deleted` upon destruction of the object that owns it.

Example A pointer member may represent a resource. A `T*` should not do so, but in older code, that's common. Consider a `T*` a possible owner and therefore suspect.

```
template<typename T>
class Smart_ptr {
    T* p;   // BAD: vague about ownership of *p
    // ...
public:
    // ... no user-defined default operations ...
};

void use(Smart_ptr<int> p1)
{
    auto p2 = p1;   // error: p2.p leaked (if not nullptr and not owned by some other
}
```

Note that if you define a destructor, you must define or delete all default operations:

```
template<typename T>
class Smart_ptr2 {
    T* p;   // BAD: vague about ownership of *p
    // ...
```

```
public:
    // ... no user-defined copy operations ...
    ~Smart_ptr2() { delete p; }  // p is an owner!
};

void use(Smart_ptr<int> p1)
{
    auto p2 = p1;   // error: double deletion
}
```

The default copy operation will just copy the `p1.p` into `p2.p` leading to a double destruction of `p1.p`. Be explicit about ownership:

```
template<typename T>
class Smart_ptr3 {
    owner<T>* p;   // OK: explicit about ownership of *p
    // ...
public:
    // ...
    // ... copy and move operations ...
    ~Smart_ptr3() { delete p; }
};

void use(Smart_ptr3<int> p1)
{
    auto p2 = p1;   // error: double deletion
}
```

Note Often the simplest way to get a destructor is to replace the pointer with a smart pointer (e.g., `std::unique_ptr`) and let the compiler arrange for proper destruction to be done implicitly.

Note Why not just require all owning pointers to be "smart pointers"? That would sometimes require non-trivial code changes and may affect ABIs.

Enforcement

- A class with a pointer data member is suspect.
- A class with an `owner<T>` should define its default operations.

C.34: If a class has an owning reference member, define a destructor

Reason A reference member may represent a resource. It should not do so, but in older code, that's common. See pointer members and destructors. Also, copying may lead to slicing.

Example, bad

```
class Handle {  // Very suspect
    Shape& s;   // use reference rather than pointer to prevent rebinding
                // BAD: vague about ownership of *p
    // ...
public:
    Handle(Shape& ss) : s{ss} { /* ... */ }
    // ...
};
```

The problem of whether `Handle` is responsible for the destruction of its `Shape` is the same as for the pointer case: If the `Handle` owns the object referred to by `s` it must have a destructor.

Example

```
class Handle {        // OK
    owner<Shape&> s;  // use reference rather than pointer to prevent rebinding
    // ...
public:
    Handle(Shape& ss) : s{ss} { /* ... */ }
    ~Handle() { delete &s; }
    // ...
};
```

Independently of whether `Handle` owns its `Shape`, we must consider the default copy operations suspect:

```
Handle x {*new Circle{p1, 17}};  // the Handle had better own the Circle or we have a l
Handle y {*new Triangle{p1, p2, p3}};
x = y;     // the default assignment will try *x.s = *y.s
```

That `x=y` is highly suspect. Assigning a `Triangle` to a `Circle`? Unless `Shape` has its copy assignment `=deleted`, only the `Shape` part of `Triangle` is copied into the `Circle`.

Note Why not just require all owning references to be replaced by "smart pointers"? Changing from references to smart pointers implies code changes. We don't (yet) have smart references. Also, that may affect ABIs.

Enforcement

- A class with a reference data member is suspect.
- A class with an `owner<T>` reference should define its default operations.

C.35: A base class destructor should be either public and virtual, or protected and nonvirtual

Reason To prevent undefined behavior. If the destructor is public, then calling code can attempt to destroy a derived class object through a base class pointer, and the result is undefined if the base class's destructor is non-virtual. If the destructor is protected, then calling code cannot destroy through a base class pointer and the destructor does not need to be virtual; it does need to be protected, not private, so that derived destructors can invoke it. In general, the writer of a base class does not know the appropriate action to be done upon destruction.

Discussion See this in the Discussion section.

Example, bad

```
struct Base {  // BAD: no virtual destructor
    virtual f();
};

struct D : Base {
    string s {"a resource needing cleanup"};
    ~D() { /* ... do some cleanup ... */ }
    // ...
};

void use()
{
    unique_ptr<Base> p = make_unique<D>();
    // ...
} // p's destruction calls ~Base(), not ~D(), which leaks D::s and possibly more
```

Note A virtual function defines an interface to derived classes that can be used without looking at the derived classes. If the interface allows destroying, it should be safe to do so.

Note A destructor must be nonprivate or it will prevent using the type :

```
class X {
    ~X();   // private destructor
    // ...
};

void use()
{
    X a;                        // error: cannot destroy
    auto p = make_unique<X>();  // error: cannot destroy
}
```

Exception We can imagine one case where you could want a protected virtual destructor: When an object of a derived type (and only of such a type) should be allowed to destroy *another* object (not itself) through a pointer to base. We haven't seen such a case in practice, though.

Enforcement

- A class with any virtual functions should have a destructor that is either public and virtual or else protected and nonvirtual.

C.36: A destructor may not fail

Reason In general we do not know how to write error-free code if a destructor should fail. The standard library requires that all classes it deals with have destructors that do not exit by throwing.

Example

```
class X {
public:
    ~X() noexcept;
    // ...
};

X::~X() noexcept
{
    // ...
    if (cannot_release_a_resource) terminate();
    // ...
}
```

Note Many have tried to devise a fool-proof scheme for dealing with failure in destructors. None have succeeded to come up with a general scheme. This can be a real practical problem: For example, what about a socket that won't close? The writer of a destructor does not know why the destructor is called and cannot "refuse to act" by throwing an exception. See discussion. To make the problem worse, many "close/release" operations are not retryable. If at all possible, consider failure to close/cleanup a fundamental design error and terminate.

Note Declare a destructor `noexcept`. That will ensure that it either completes normally or terminate the program.

Note If a resource cannot be released and the program may not fail, try to signal the failure to the rest of the system somehow (maybe even by modifying some global state and hope something will notice and be able to take care of the problem). Be fully aware that this technique is special-purpose and error-prone. Consider the "my connection will not close" example. Probably there is a problem at the other end of the connection and only a piece of code responsible for both ends of the connection can properly handle the problem. The destructor could send a message (somehow) to the responsible part of the system, consider that to have closed the connection, and return normally.

Note If a destructor uses operations that may fail, it can catch exceptions and in some cases still complete successfully (e.g., by using a different clean-up mechanism from the one that threw an exception).

Enforcement (Simple) A destructor should be declared `noexcept`.

C.37: Make destructors `noexcept`

Reason A destructor may not fail. If a destructor tries to exit with an exception, it's a bad design error and the program had better terminate.

Note A destructor (either user-defined or compiler-generated) is implicitly declared `noexcept` (independently of what code is in its body) if all of the members of its class have `noexcept` destructors.

Enforcement (Simple) A destructor should be declared `noexcept`.

C.ctor: Constructors

A constructor defines how an object is initialized (constructed).

C.40: Define a constructor if a class has an invariant

Reason That's what constructors are for.

Example

```
class Date {  // a Date represents a valid date
              // in the January 1, 1900 to December 31, 2100 range
    Date(int dd, int mm, int yy)
        :d{dd}, m{mm}, y{yy}
    {
        if (!is_valid(d, m, y)) throw Bad_date{};  // enforce invariant
    }
    // ...
private:
    int d, m, y;
};
```

It is often a good idea to express the invariant as an `Ensures` on the constructor.

Note A constructor can be used for convenience even if a class does not have an invariant. For example:

```
struct Rec {
    string s;
    int i {0};
    Rec(const string& ss) : s{ss} {}
    Rec(int ii) :i{ii} {}
};

Rec r1 {7};
Rec r2 {"Foo bar"};
```

Note The C++11 initializer list rule eliminates the need for many constructors. For example:

```
struct Rec2{
    string s;
    int i;
    Rec2(const string& ss, int ii = 0) :s{ss}, i{ii} {}   // redundant
};

Rec r1 {"Foo", 7};
Rec r2 {"Bar"};
```

The `Rec2` constructor is redundant. Also, the default for `int` would be better done as a member initializer.

See also: construct valid object and constructor throws.

Enforcement

- Flag classes with user-defined copy operations but no constructor (a user-defined copy is a good indicator that the class has an invariant)

C.41: A constructor should create a fully initialized object

Reason A constructor establishes the invariant for a class. A user of a class should be able to assume that a constructed object is usable.

Example, bad

```
class X1 {
    FILE* f;   // call init() before any other function
    // ...
public:
    X1() {}
    void init();   // initialize f
    void read();   // read from f
    // ...
};

void f()
{
    X1 file;
    file.read();   // crash or bad read!
    // ...
    file.init();   // too late
    // ...
}
```

Compilers do not read comments.

Exception: If a valid object cannot conveniently be constructed by a constructor use a factory function.

Note If a constructor acquires a resource (to create a valid object), that resource should be released by the destructor. The idiom of having constructors acquire resources and destructors release them is called RAII ("Resource Acquisition Is Initialization").

C.42: If a constructor cannot construct a valid object, throw an exception

Reason Leaving behind an invalid object is asking for trouble.

Example

```
class X2 {
    FILE* f;   // call init() before any other function
    // ...
public:
    X2(const string& name)
        :f{fopen(name.c_str(), "r")}
    {
        if (f == nullptr) throw runtime_error{"could not open" + name};
        // ...
    }

    void read();      // read from f
    // ...
};

void f()
{
    X2 file {"Zeno"}; // throws if file isn't open
    file.read();      // fine
    // ...
}
```

Example, bad

```
class X3 {     // bad: the constructor leaves a non-valid object behind
    FILE* f;   // call init() before any other function
    bool valid;
    // ...
public:
    X3(const string& name)
        :f{fopen(name.c_str(), "r")}, valid{false}
    {
        if (f) valid = true;
        // ...
    }

    void is_valid() { return valid; }
    void read();   // read from f
```

```
    // ...
};

void f()
{
    X3 file {"Heraclides"};
    file.read();   // crash or bad read!
    // ...
    if (is_valid()) {
        file.read();
        // ...
    }
    else {
        // ... handle error ...
    }
    // ...
}
```

Note For a variable definition (e.g., on the stack or as a member of another object) there is no explicit function call from which an error code could be returned. Leaving behind an invalid object and relying on users to consistently check an `is_valid()` function before use is tedious, error-prone, and inefficient.

Exception: There are domains, such as some hard-real-time systems (think airplane controls) where (without additional tool support) exception handling is not sufficiently predictable from a timing perspective. There the `is_valid()` technique must be used. In such cases, check `is_valid()` consistently and immediately to simulate RAII.

Alternative: If you feel tempted to use some "post-constructor initialization" or "two-stage initialization" idiom, try not to do that. If you really have to, look at factory functions.

Note One reason people have used `init()` functions rather than doing the initialization work in a constructor has been to avoid code replication. Delegating constructors and default member initialization do that better. Another reason is been to delay initialization until an object is needed; the solution to that is often not to declare a variable until it can be properly initialized

Enforcement

- (Simple) Every constructor should initialize every member variable (either explicitly, via a delegating ctor call or via default construction).
- (Unknown) If a constructor has an `Ensures` contract, try to see if it holds as a postcondition.

C.43: Ensure that a class has a default constructor

Reason Many language and library facilities rely on default constructors to initialize their elements, e.g. `T a[10]` and `std::vector<T> v(10)`.

Example , bad

```
class Date { // BAD: no default constructor
public:
    Date(int dd, int mm, int yyyy);
    // ...
};

vector<Date> vd1(1000);   // default Date needed here
vector<Date> vd2(1000, Date{Month::october, 7, 1885});   // alternative
```

The default constructor is only auto-generated if there is no user-declared constructor, hence it's impossible to initialize the vector `vd1` in the example above.

There is no "natural" default date (the big bang is too far back in time to be useful for most people), so this example is non-trivial. `{0, 0, 0}` is not a valid date in most calendar systems, so choosing that would be introducing something like floating-point's `NaN`. However, most realistic `Date` classes have a "first date" (e.g. January 1, 1970 is popular), so making that the default is usually trivial.

Example

```
class Date {
public:
    Date(int dd, int mm, int yyyy);
    Date() = default; // See also C.45
    // ...
private:
    int dd = 1;
    int mm = 1;
    int yyyy = 1970;
    // ...
};

vector<Date> vd1(1000);
```

Note A class with members that all have default constructors implicitly gets a default constructor:

```
struct X {
    string s;
    vector v;
};

X x; // means X{ {}, {} }; that is the empty string and the empty vector
```

Beware that built-in types are not properly default constructed:

```
struct X {
   string s;
   int i;
};

void f()
{
   X x;    // x.s is initialized to the empty string; x.i is uninitialized

   cout << x.s << ' ' << x.i << '\n';
   ++x.i;
}
```

Statically allocated objects of built-in types are by default initialized to `0`, but local built-in variables are not. Beware that your compiler may default initialize local built-in variables, whereas an optimized build will not. Thus, code like the example above may appear to work, but it relies on undefined behavior. Assuming that you want initialization, an explicit default initialization can help:

```
struct X {
   string s;
   int i {};   // default initialize (to 0)
};
```

Enforcement

- Flag classes without a default constructor

C.44: Prefer default constructors to be simple and non-throwing

Reason Being able to set a value to "the default" without operations that might fail simplifies error handling and reasoning about move operations.

Example, problematic

```
template<typename T>
class Vector0 {   // elem points to space-elem element allocated using new
public:
    Vector0() :Vector0{0} {}
    Vector0(int n) :elem{new T[n]}, space{elem + n}, last{elem} {}
    // ...
private:
    own<T*> elem;
    T* space;
    T* last;
};
```

This is nice and general, but setting a `Vector0` to empty after an error involves an allocation, which may fail. Also, having a default `Vector` represented as `{new T[0], 0, 0}` seems wasteful. For example, `Vector0 v(100)` costs 100 allocations.

Example

```
template<typename T>
class Vector1 {   // elem is nullptr or elem points to space-elem element allocated us
public:
    Vector1() noexcept {}   // sets the representation to {nullptr, nullptr, nullptr};
    Vector1(int n) :elem{new T[n]}, space{elem + n}, last{elem} {}
    // ...
private:
    own<T*> elem = nullptr;
    T* space = nullptr;
    T* last = nullptr;
};
```

Using `{nullptr, nullptr, nullptr}` makes `Vector1{}` cheap, but a special case and implies run-time checks. Setting a `Vector1` to empty after detecting an error is trivial.

Enforcement

- Flag throwing default constructors

C.45: Don't define a default constructor that only initializes data members; use in-class member initializers instead

Reason Using in-class member initializers lets the compiler generate the function for you. The compiler-generated function can be more efficient.

Example, bad

```
class X1 { // BAD: doesn't use member initializers
    string s;
    int i;
public:
    X1() :s{"default"}, i{1} { }
    // ...
};
```

Example

```
class X2 {
    string s = "default";
    int i = 1;
public:
    // use compiler-generated default constructor
    // ...
};
```

Enforcement (Simple) A default constructor should do more than just initialize member variables with constants.

C.46: By default, declare single-argument constructors explicit

Reason To avoid unintended conversions.

Example, bad

```
class String {
    // ...
public:
    String(int);   // BAD
    // ...
};

String s = 10;   // surprise: string of size 10
```

Exception If you really want an implicit conversion from the constructor argument type to the class type, don't use `explicit`:

```
class Complex {
    // ...
public:
    Complex(double d);   // OK: we want a conversion from d to {d, 0}
    // ...
};

Complex z = 10.7;   // unsurprising conversion
```

See also: Discussion of implicit conversions.

Enforcement (Simple) Single-argument constructors should be declared `explicit`. Good single argument non-`explicit` constructors are rare in most code based. Warn for all that are not on a "positive list".

C.47: Define and initialize member variables in the order of member declaration

Reason To minimize confusion and errors. That is the order in which the initialization happens (independent of the order of member initializers).

Example, bad

```
class Foo {
    int m1;
    int m2;
public:
    Foo(int x) :m2{x}, m1{++x} { }   // BAD: misleading initializer order
    // ...
};

Foo x(1); // surprise: x.m1 == x.m2 == 2
```

Enforcement (Simple) A member initializer list should mention the members in the same order they are declared.

See also: Discussion

C.48: Prefer in-class initializers to member initializers in constructors for constant initializers

Reason Makes it explicit that the same value is expected to be used in all constructors. Avoids repetition. Avoids maintenance problems. It leads to the shortest and most efficient code.

Example, bad

```
class X {   // BAD
    int i;
    string s;
    int j;
public:
    X() :i{666}, s{"qqq"} { }   // j is uninitialized
    X(int ii) :i{ii} {}         // s is "" and j is uninitialized
    // ...
};
```

How would a maintainer know whether `j` was deliberately uninitialized (probably a poor idea anyway) and whether it was intentional to give `s` the default value `""` in one case and `qqq` in another (almost certainly a bug)? The problem with `j` (forgetting to initialize a member) often happens when a new member is added to an existing class.

Example

```
class X2 {
    int i {666};
    string s {"qqq"};
    int j {0};
public:
    X2() = default;        // all members are initialized to their defaults
    X2(int ii) :i{ii} {}   // s and j initialized to their defaults
    // ...
};
```

Alternative: We can get part of the benefits from default arguments to constructors, and that is not uncommon in older code. However, that is less explicit, causes more arguments to be passed, and is repetitive when there is more than one constructor:

```
class X3 {   // BAD: inexplicit, argument passing overhead
    int i;
    string s;
    int j;
public:
    X3(int ii = 666, const string& ss = "qqq", int jj = 0)
        :i{ii}, s{ss}, j{jj} { }   // all members are initialized to their defaults
    // ...
};
```

Enforcement

- (Simple) Every constructor should initialize every member variable (either explicitly, via a delegating ctor call or via default construction).
- (Simple) Default arguments to constructors suggest an in-class initializer may be more appropriate.

C.49: Prefer initialization to assignment in constructors

Reason An initialization explicitly states that initialization, rather than assignment, is done and can be more elegant and efficient. Prevents "use before set" errors.

Example, good

```
class A {   // Good
    string s1;
public:
    A() : s1{"Hello, "} { }    // GOOD: directly construct
    // ...
};
```

Example, bad

```
class B {   // BAD
    string s1;
public:
    B() { s1 = "Hello, "; }   // BAD: default constructor followed by assignment
    // ...
};

class C {   // UGLY, aka very bad
    int* p;
public:
    C() { cout << *p; p = new int{10}; }   // accidental use before initialized
    // ...
};
```

C.50: Use a factory function if you need "virtual behavior" during initialization

Reason If the state of a base class object must depend on the state of a derived part of the object, we need to use a virtual function (or equivalent) while minimizing the window of opportunity to misuse an imperfectly constructed object.

Example, bad

```
class B {
public:
    B()
    {
        // ...
        f();   // BAD: virtual call in constructor
        // ...
    }

    virtual void f() = 0;

    // ...
};
```

Example

```
class B {
protected:
    B() { /* ... */ }                     // create an imperfectly initialized object

    virtual void PostInitialize()       // to be called right after construction
    {
        // ...
        f();    // GOOD: virtual dispatch is safe
        // ...
    }

public:
    virtual void f() = 0;

    template<class T>
    static shared_ptr<T> Create()    // interface for creating objects
    {
        auto p = make_shared<T>();
        p->PostInitialize();
        return p;
    }
};

class D : public B { /* "¦ */ };            // some derived class

shared_ptr<D> p = D::Create<D>();        // creating a D object
```

By making the constructor `protected` we avoid an incompletely constructed object escaping into the wild. By providing the factory function `Create()`, we make construction (on the free store) convenient.

Note Conventional factory functions allocate on the free store, rather than on the stack or in an enclosing object.

See also: Discussion

C.51: Use delegating constructors to represent common actions for all constructors of a class

Reason To avoid repetition and accidental differences.

Example, bad

```
class Date {   // BAD: repetitive
    int d;
    Month m;
    int y;
public:
    Date(int ii, Month mm, year yy)
        :i{ii}, m{mm} y{yy}
        { if (!valid(i, m, y)) throw Bad_date{}; }

    Date(int ii, Month mm)
        :i{ii}, m{mm} y{current_year()}
        { if (!valid(i, m, y)) throw Bad_date{}; }
    // ...
};
```

The common action gets tedious to write and may accidentally not be common.

Example

```
class Date2 {
    int d;
    Month m;
    int y;
public:
    Date2(int ii, Month mm, year yy)
        :i{ii}, m{mm} y{yy}
        { if (!valid(i, m, y)) throw Bad_date{}; }
```

```
    Date2(int ii, Month mm)
        :Date2{ii, mm, current_year()} {}
    // ...
};
```

See also: If the "repeated action" is a simple initialization, consider an in-class member initializer.

Enforcement (Moderate) Look for similar constructor bodies.

C.52: Use inheriting constructors to import constructors into a derived class that does not need further explicit initialization

Reason If you need those constructors for a derived class, re-implementing them is tedious and error prone.

Example `std::vector` has a lot of tricky constructors, so if I want my own `vector`, I don't want to reimplement them:

```
class Rec {
    // ... data and lots of nice constructors ...
};

class Oper : public Rec {
    using Rec::Rec;
    // ... no data members ...
    // ... lots of nice utility functions ...
};
```

Example, bad

```
struct Rec2 : public Rec {
    int x;
    using Rec::Rec;
};

Rec2 r {"foo", 7};
int val = r.x;    // uninitialized
```

Enforcement Make sure that every member of the derived class is initialized.

C.copy: Copy and move

Value types should generally be copyable, but interfaces in a class hierarchy should not. Resource handles may or may not be copyable. Types can be defined to move for logical as well as performance reasons.

C.60: Make copy assignment non-`virtual`, take the parameter by `const&`, and return by non-`const&`

Reason It is simple and efficient. If you want to optimize for rvalues, provide an overload that takes a `&&` (see F.24).

Example

```
class Foo {
public:
    Foo& operator=(const Foo& x)
    {
        auto tmp = x;    // GOOD: no need to check for self-assignment (other than perf
        std::swap(*this, tmp);
        return *this;
    }
    // ...
};

Foo a;
Foo b;
Foo f();

a = b;    // assign lvalue: copy
a = f();  // assign rvalue: potentially move
```

Note The `swap` implementation technique offers the strong guarantee.

Example But what if you can get significantly better performance by not making a temporary copy? Consider a simple `Vector` intended for a domain where assignment of large, equal-sized `Vector`s is common. In this case, the copy of elements implied by the `swap` implementation technique could cause an order of magnitude increase in cost:

```
template<typename T>
class Vector {
public:
```

```
    Vector& operator=(const Vector&);
    // ...
private:
    T* elem;
    int sz;
};

Vector& Vector::operator=(const Vector& a)
{
    if (a.sz > sz) {
        // ... use the swap technique, it can't be bettered ...
        return *this
    }
    // ... copy sz elements from *a.elem to elem ...
    if (a.sz < sz) {
        // ... destroy the surplus elements in *this* and adjust size ...
    }
    return *this;
}
```

By writing directly to the target elements, we will get only the basic guarantee rather than the strong guarantee offered by the `swap` technique. Beware of self assignment.

Alternatives: If you think you need a `virtual` assignment operator, and understand why that's deeply problematic, don't call it `operator=`. Make it a named function like `virtual void assign(const Foo&)`. See copy constructor vs. `clone()`.

Enforcement

- (Simple) An assignment operator should not be virtual. Here be dragons!
- (Simple) An assignment operator should return `T&` to enable chaining, not alternatives like `const T&` which interfere with composability and putting objects in containers.
- (Moderate) An assignment operator should (implicitly or explicitly) invoke all base and member assignment operators. Look at the destructor to determine if the type has pointer semantics or value semantics.

C.61: A copy operation should copy

Reason That is the generally assumed semantics. After `x=y`, we should have `x == y`. After a copy `x` and `y` can be independent objects (value semantics, the way non-pointer built-in types and the standard-library types work) or refer to a shared object (pointer semantics, the way pointers work).

Example

```
class X {   // OK: value semantics
public:
    X();
    X(const X&);     // copy X
    void modify();   // change the value of X
    // ...
    ~X() { delete[] p; }
private:
    T* p;
    int sz;
};

bool operator==(const X& a, const X& b)
{
    return a.sz == b.sz && equal(a.p, a.p + a.sz, b.p, b.p + b.sz);
}

X::X(const X& a)
    :p{new T[a.sz]}, sz{a.sz}
{
    copy(a.p, a.p + sz, a.p);
}

X x;
X y = x;
if (x != y) throw Bad{};
x.modify();
if (x == y) throw Bad{};   // assume value semantics
```

Example

```
class X2 {  // OK: pointer semantics
public:
    X2();
    X2(const X&) = default; // shallow copy
    ~X2() = default;
    void modify();          // change the value of X
    // ...
private:
    T* p;
    int sz;
};
```

```
bool operator==(const X2& a, const X2& b)
{
    return a.sz == b.sz && a.p == b.p;
}

X2 x;
X2 y = x;
if (x != y) throw Bad{};
x.modify();
if (x != y) throw Bad{};  // assume pointer semantics
```

Note Prefer copy semantics unless you are building a "smart pointer". Value semantics is the simplest to reason about and what the standard library facilities expect.

Enforcement (Not enforceable)

C.62: Make copy assignment safe for self-assignment

Reason If `x=x` changes the value of `x`, people will be surprised and bad errors will occur (often including leaks).

Example The standard-library containers handle self-assignment elegantly and efficiently:

```
std::vector<int> v = {3, 1, 4, 1, 5, 9};
v = v;
// the value of v is still {3, 1, 4, 1, 5, 9}
```

Note The default assignment generated from members that handle self-assignment correctly handles self-assignment.

```
struct Bar {
    vector<pair<int, int>> v;
    map<string, int> m;
    string s;
};

Bar b;
// ...
b = b;   // correct and efficient
```

Note You can handle self-assignment by explicitly testing for self-assignment, but often it is faster and more elegant to cope without such a test (e.g., using `swap`).

```
class Foo {
    string s;
    int i;
public:
    Foo& operator=(const Foo& a);
    // ...
};

Foo& Foo::operator=(const Foo& a)   // OK, but there is a cost
{
    if (this == &a) return *this;
    s = a.s;
    i = a.i;
    return *this;
}
```

This is obviously safe and apparently efficient. However, what if we do one self-assignment per million assignments? That's about a million redundant tests (but since the answer is essentially always the same, the computer's branch predictor will guess right essentially every time). Consider:

```
Foo& Foo::operator=(const Foo& a)   // simpler, and probably much better
{
    s = a.s;
    i = a.i;
    return *this;
}
```

`std::string` is safe for self-assignment and so are `int`. All the cost is carried by the (rare) case of self-assignment.

Enforcement (Simple) Assignment operators should not contain the pattern `if (this == &a) return *this;` ???

C.63: Make move assignment non-`virtual`, take the parameter by `&&`, and return by non-`const &`

Reason It is simple and efficient.

See: The rule for copy-assignment.

Enforcement Equivalent to what is done for copy-assignment.

- (Simple) An assignment operator should not be virtual. Here be dragons!
- (Simple) An assignment operator should return `T&` to enable chaining, not alternatives like `const T&` which interfere with composability and putting objects in containers.
- (Moderate) A move assignment operator should (implicitly or explicitly) invoke all base and member move assignment operators.

C.64: A move operation should move and leave its source in valid state

Reason That is the generally assumed semantics. After `x=std::move(y)` the value of `x` should be the value `y` had and `y` should be in a valid state.

Example

```
template<typename T>
class X {   // OK: value semantics
public:
    X();
    X(X&& a);           // move X
    void modify();      // change the value of X
    // ...
    ~X() { delete[] p; }
private:
    T* p;
    int sz;
};


X::X(X&& a)
    :p{a.p}, sz{a.sz}  // steal representation
{
    a.p = nullptr;      // set to "empty"
    a.sz = 0;
}

void use()
{
    X x{};
    // ...
    X y = std::move(x);
    x = X{};   // OK
} // OK: x can be destroyed
```

Note Ideally, that moved-from should be the default value of the type. Ensure that unless there is an exceptionally good reason not to. However, not all types have a default value and for some types establishing the default value can be expensive. The standard requires only that the moved-from object can be destroyed. Often, we can easily and cheaply do better: The standard library assumes that it it possible to assign to a moved-from object. Always leave the moved-from object in some (necessarily specified) valid state.

Note Unless there is an exceptionally strong reason not to, make `x = std::move(y); y = z;` work with the conventional semantics.

Enforcement (Not enforceable) Look for assignments to members in the move operation. If there is a default constructor, compare those assignments to the initializations in the default constructor.

C.65: Make move assignment safe for self-assignment

Reason If `x = x` changes the value of `x`, people will be surprised and bad errors may occur. However, people don't usually directly write a self-assignment that turn into a move, but it can occur. However, `std::swap` is implemented using move operations so if you accidentally do `swap(a, b)` where `a` and `b` refer to the same object, failing to handle self-move could be a serious and subtle error.

Example

```
class Foo {
    string s;
    int i;
public:
    Foo& operator=(Foo&& a);
    // ...
};

Foo& Foo::operator=(Foo&& a)       // OK, but there is a cost
{
    if (this == &a) return *this;  // this line is redundant
    s = std::move(a.s);
    i = a.i;
    return *this;
}
```

The one-in-a-million argument against `if (this == &a) return *this;` tests from the discussion of self-assignment is even more relevant for self-move.

Note There is no know general way of avoiding a `if (this == &a) return *this;` test for a move assignment and still get a correct answer (i.e., after `x=x` the value of `x` is unchanged).

Note The ISO standard guarantees only a "valid but unspecified" state for the standard library containers. Apparently this has not been a problem in about 10 years of experimental and production use. Please contact the editors if you find a counter example. The rule here is more caution and insists on complete safety.

Example Here is a way to move a pointer without a test (imagine it as code in the implementation a move assignment):

```
// move from other.ptr to this->ptr
T* temp = other.ptr;
other.ptr = nullptr;
delete ptr;
ptr = temp;
```

Enforcement

- (Moderate) In the case of self-assignment, a move assignment operator should not leave the object holding pointer members that have been `delete`d or set to nullptr.
- (Not enforceable) Look at the use of standard-library container types (incl. `string`) and consider them safe for ordinary (not life-critical) uses.

C.66: Make move operations `noexcept`

Reason A throwing move violates most people's reasonably assumptions. A non-throwing move will be used more efficiently by standard-library and language facilities.

Example

```
template<typename T>
class Vector {
    // ...
    Vector(Vector&& a) noexcept :elem{a.elem}, sz{a.sz} { a.sz = 0; a.elem = nullptr; }
    Vector& operator=(Vector&& a) noexcept { elem = a.elem; sz = a.sz; a.sz = 0; a.elem
    // ...
public:
    T* elem;
    int sz;
};
```

These copy operations do not throw.

Example, bad

```
template<typename T>
class Vector2 {
    // ...
    Vector2(Vector2&& a) { *this = a; }             // just use the copy
    Vector2& operator=(Vector2&& a) { *this = a; }  // just use the copy
    // ...
public:
    T* elem;
    int sz;
};
```

This `Vector2` is not just inefficient, but since a vector copy requires allocation, it can throw.

Enforcement (Simple) A move operation should be marked `noexcept`.

C.67: A base class should suppress copying, and provide a virtual `clone` instead if "copying" is desired

Reason To prevent slicing, because the normal copy operations will copy only the base portion of a derived object.

Example, bad

```
class B { // BAD: base class doesn't suppress copying
    int data;
    // ... nothing about copy operations, so uses default ...
};

class D : public B {
    string moredata; // add a data member
    // ...
};

auto d = make_unique<D>();
auto b = make_unique<B>(d); // oops, slices the object; gets only d.data but drops d.mo
```

Example

```
class B { // GOOD: base class suppresses copying
    B(const B&) =delete;
```

```
    B& operator=(const B&) =delete;
    virtual unique_ptr<B> clone() { return /* B object */; }
    // ...
};

class D : public B {
    string moredata; // add a data member
    unique_ptr<B> clone() override { return /* D object */; }
    // ...
};

auto d = make_unique<D>();
auto b = d.clone(); // ok, deep clone
```

Note It's good to return a smart pointer, but unlike with raw pointers the return type cannot be covariant (for example, `D::clone` can't return a `unique_ptr<D>`. Don't let this tempt you into returning an owning raw pointer; this is a minor drawback compared to the major robustness benefit delivered by the owning smart pointer.

Exceptions If you need covariant return types, return an `owner<derived*>`. See C.130.

Enforcement A class with any virtual function should not have a copy constructor or copy assignment operator (compiler-generated or handwritten).

C.other: Other default operation rules

In addition to the operations for which the language offer default implementations, there are a few operations that are so foundational that it rules for their definition are needed: comparisons, `swap`, and `hash`.

C.80: Use `=default` if you have to be explicit about using the default semantics

Reason The compiler is more likely to get the default semantics right and you cannot implement these function better than the compiler.

Example

```
class Tracer {
    string message;
public:
    Tracer(const string& m) : message{m} { cerr << "entering " << message << '\n'; }
```

```
    ~Tracer() { cerr << "exiting " << message << '\n'; }

    Tracer(const Tracer&) = default;
    Tracer& operator=(const Tracer&) = default;
    Tracer(Tracer&&) = default;
    Tracer& operator=(Tracer&&) = default;
};
```

Because we defined the destructor, we must define the copy and move operations. The `=default` is the best and simplest way of doing that.

Example, bad

```
class Tracer2 {
    string message;
public:
    Tracer2(const string& m) : message{m} { cerr << "entering " << message << '\n'; }
    ~Tracer2() { cerr << "exiting " << message << '\n'; }

    Tracer2(const Tracer2& a) : message{a.message} {}
    Tracer2& operator=(const Tracer2& a) { message = a.message; }
    Tracer2(Tracer2&& a) :message{a.message} {}
    Tracer2& operator=(Tracer2&& a) { message = a.message; }
};
```

Writing out the bodies of the copy and move operations is verbose, tedious, and error-prone. A compiler does it better.

Enforcement (Moderate) The body of a special operation should not have the same accessibility and semantics as the compiler-generated version, because that would be redundant

C.81: Use `=delete` when you want to disable default behavior (without wanting an alternative)

Reason In a few cases, a default operation is not desirable.

Example

```
class Immortal {
public:
    ~Immortal() = delete;    // do not allow destruction
    // ...
```

```
};

void use()
{
    Immortal ugh;   // error: ugh cannot be destroyed
    Immortal* p = new Immortal{};
    delete p;       // error: cannot destroy *p
}
```

Example A `unique_ptr` can be moved, but not copied. To achieve that its copy operations are deleted. To avoid copying it is necessary to `=delete` its copy operations from lvalues:

```
template <class T, class D = default_delete<T>> class unique_ptr {
public:
    // ...
    constexpr unique_ptr() noexcept;
    explicit unique_ptr(pointer p) noexcept;
    // ...
    unique_ptr(unique_ptr&& u) noexcept;   // move constructor
    // ...
    unique_ptr(const unique_ptr&) = delete; // disable copy from lvalue
    // ...
};

unique_ptr<int> make();   // make "something" and return it by moving

void f()
{
    unique_ptr<int> pi {};
    auto pi2 {pi};      // error: no move constructor from lvalue
    auto pi3 {make()};  // OK, move: the result of make() is an rvalue
}
```

Enforcement The elimination of a default operation is (should be) based on the desired semantics of the class. Consider such classes suspect, but maintain a "positive list" of classes where a human has asserted that the semantics is correct.

C.82: Don't call virtual functions in constructors and destructors

Reason The function called will be that of the object constructed so far, rather than a possibly overriding function in a derived class. This can be most confusing. Worse, a direct or indirect call to an unimplemented pure virtual function from a constructor or destructor results in undefined behavior.

Example, bad

```
class base {
public:
    virtual void f() = 0;   // not implemented
    virtual void g();       // implemented with base version
    virtual void h();       // implemented with base version
};

class derived : public base {
public:
    void g() override;      // provide derived implementation
    void h() final;         // provide derived implementation

    derived()
    {
        f();                // BAD: attempt to call an unimplemented virtual function

        g();                // BAD: will call derived::g, not dispatch further virtuall
        derived::g();       // GOOD: explicitly state intent to call only the visible v

        h();                // ok, no qualification needed, h is final
    }
};
```

Note that calling a specific explicitly qualified function is not a virtual call even if the function is `virtual`.

See also factory functions for how to achieve the effect of a call to a derived class function without risking undefined behavior.

Note There is nothing inherently wrong with calling virtual functions constructors and destructors. The semantics of such calls is type safe. However, experience shows that such calls are rarely needed, easily confuse maintainers, and become a source of errors when used by novices.

Enforcement

- Flag calls of virtual functions from constructors and destructors.

C.83: For value-like types, consider providing a `noexcept` swap function

Reason A `swap` 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. Consider using swap to implement copy assignment in terms of copy construction. See also destructors, deallocation, and swap must never fail.

Example, good

```
class Foo {
    // ...
public:
    void swap(Foo& rhs) noexcept
    {
        m1.swap(rhs.m1);
        std::swap(m2, rhs.m2);
    }
private:
    Bar m1;
    int m2;
};
```

Providing a nonmember `swap` function in the same namespace as your type for callers' convenience.

```
void swap(Foo& a, Foo& b)
{
    a.swap(b);
}
```

Enforcement

- (Simple) A class without virtual functions should have a `swap` member function declared.
- (Simple) When a class has a `swap` member function, it should be declared `noexcept`.

C.84: A `swap` function may not fail

Reason `swap` is widely used in ways that are assumed never to fail and programs cannot easily be written to work correctly in the presence of a failing `swap`. The standard-library containers and algorithms will not work correctly if a swap of an element type fails.

Example, bad

```
void swap(My_vector& x, My_vector& y)
{
    auto tmp = x;   // copy elements
    x = y;
    y = tmp;
}
```

This is not just slow, but if a memory allocation occurs for the elements in `tmp`, this `swap` may throw and would make STL algorithms fail if used with them.

Enforcement (Simple) When a class has a `swap` member function, it should be declared `noexcept`.

C.85: Make `swap` `noexcept`

Reason A `swap` may not fail. If a `swap` tries to exit with an exception, it's a bad design error and the program had better terminate.

Enforcement (Simple) When a class has a `swap` member function, it should be declared `noexcept`.

C.86: Make `==` symmetric with respect to operand types and `noexcept`

Reason Asymmetric treatment of operands is surprising and a source of errors where conversions are possible. `==` is a fundamental operations and programmers should be able to use it without fear of failure.

Example

```
class X {
    string name;
    int number;
};

bool operator==(const X& a, const X& b) noexcept { return a.name == b.name && a.number =
```

Example, bad

```
class B {
    string name;
    int number;
    bool operator==(const B& a) const { return name == a.name && number == a.number; }
    // ...
};
```

`B`'s comparison accepts conversions for its second operand, but not its first.

Note If a class has a failure state, like `double`'s `NaN`, there is a temptation to make a comparison against the failure state throw. The alternative is to make two failure states compare equal and any valid state compare false against the failure state.

Note This rule applies to all the usual comparison operators: `!=',`<, `<=`, `>`, and `>=`.

Enforcement

- Flag an `operator==()` for which the argument types differ; same for other comparison operators: `!=',`<, `<=`, `>`, and `>=`.
- Flag member `operator==()`s; same for other comparison operators: `!=',`<, `<=`, `>`, and `>=`.

C.87: Beware of `==` on base classes

Reason It is really hard to write a foolproof and useful `==` for a hierarchy.

Example, bad

```
class B {
    string name;
    int number;
    virtual bool operator==(const B& a) const
    {
        return name == a.name && number == a.number;
    }
    // ...
};
```

`B`'s comparison accepts conversions for its second operand, but not its first.

```
class D :B {
    char character;
    virtual bool operator==(const D& a) const
    {
        return name == a.name && number == a.number && character == a.character;
    }
    // ...
};

B b = ...
D d = ...
b == d;    // compares name and number, ignores d's character
d == b;    // error: no == defined
D d2;
d == d2;   // compares name, number, and character
B& b2 = d2;
b2 == d;   // compares name and number, ignores d2's and d's character
```

Of course there are ways of making `==` work in a hierarchy, but the naive approaches do not scale

Note This rule applies to all the usual comparison operators: `!=',<, <=, >, and >=.

Enforcement

- Flag a virtual `operator==()`; same for other comparison operators: `!=',<, <=, >, and >=.

C.89: Make a `hash` `noexcept`

Reason Users of hashed containers use hash indirectly and don't expect simple access to throw. It's a standard-library requirement.

Example, bad

```
template<>
struct hash<My_type> {  // thoroughly bad hash specialization
   using result_type = size_t;
   using argument_type = My_type;

   size_t operator() (const My_type & x) const
   {
```

```
        size_t xs = x.s.size();
        if (xs < 4) throw Bad_My_type{};    // "Nobody expects the Spanish inquisition!"
        return hash<size_t>()(x.s.size()) ^ trim(x.s);
    }
};

int main()
{
     unordered_map<My_type,int> m;
    My_type mt{ "asdfg" };
    m[mt] = 7;
    cout << m[My_type{ "asdfg" }] << '\n';
}
```

If you have to define a `hash` specialization, try simply to let it combine standard-library `hash` specializations with `^` (xor). That tends to work better than "cleverness" for non-specialists.

Enforcement

- Flag throwing `hash`es.

C.con: Containers and other resource handles

A container is an object holding a sequence of objects of some type; `std::vector` is the archetypical container. A resource handle is a class that owns a resource; `std::vector` is the typical resource handle; its resource is its sequence of elements.

Summary of container rules:

- C.100: Follow the STL when defining a container
- C.101: Give a container value semantics
- C.102: Give a container move operations
- C.103: Give a container an initializer list constructor
- C.104: Give a container a default constructor that sets it to empty
- C.105: Give a constructor and `Extent` constructor
- ???
- C.109: If a resource handle has pointer semantics, provide `*` and `->`

See also: Resources

C.lambdas: Function objects and lambdas

A function object is an object supplying an overloaded `()` so that you can call it. A lambda expression (colloquially often shortened to "a lambda") is a notation for generating a function object. Function objects should be cheap to copy (and therefore passed by value).

Summary:

- F.50: Use a lambda when a function won't do (to capture local variables, or to write a local function)
- F.52: Prefer capturing by reference in lambdas that will be used locally, including passed to algorithms
- F.53: Avoid capturing by reference in lambdas that will be used nonlocally, including returned, stored on the heap, or passed to another thread
- ES.28: Use lambdas for complex initialization, especially of `const` variables

C.hier: Class hierarchies (OOP)

A class hierarchy is constructed to represent a set of hierarchically organized concepts (only). Typically base classes act as interfaces. There are two major uses for hierarchies, often named implementation inheritance and interface inheritance.

Class hierarchy rule summary:

- C.120: Use class hierarchies to represent concepts with inherent hierarchical structure
- C.121: If a base class is used as an interface, make it a pure abstract class
- C.122: Use abstract classes as interfaces when complete separation of interface and implementation is needed

Designing rules for classes in a hierarchy summary:

- C.126: An abstract class typically doesn't need a constructor
- C.127: A class with a virtual function should have a virtual or protected destructor
- C.128: Use `override` to make overriding explicit in large class hierarchies
- C.129: When designing a class hierarchy, distinguish between implementation inheritance and interface inheritance
- C.130: Redefine or prohibit copying for a base class; prefer a virtual `clone` function instead
- C.131: Avoid trivial getters and setters
- C.132: Don't make a function `virtual` without reason
- C.133: Avoid `protected` data
- C.134: Ensure all non-`const` data members have the same access level
- C.135: Use multiple inheritance to represent multiple distinct interfaces

- C.136: Use multiple inheritance to represent the union of implementation attributes
- C.137: Use `virtual` bases to avoid overly general base classes
- C.138: Create an overload set for a derived class and its bases with `using`
- C.139: Use `final` sparingly
- C.140: Do not provide different default arguments for a virtual function and an overrider

Accessing objects in a hierarchy rule summary:

- C.145: Access polymorphic objects through pointers and references
- C.146: Use `dynamic_cast` where class hierarchy navigation is unavoidable
- C.147: Use `dynamic_cast` to a reference type when failure to find the required class is considered an error
- C.148: Use `dynamic_cast` to a pointer type when failure to find the required class is considered a valid alternative
- C.149: Use `unique_ptr` or `shared_ptr` to avoid forgetting to `delete` objects created using `new`
- C.150: Use `make_unique()` to construct objects owned by `unique_ptr`s
- C.151: Use `make_shared()` to construct objects owned by `shared_ptr`s
- C.152: Never assign a pointer to an array of derived class objects to a pointer to its base

C.120: Use class hierarchies to represent concepts with inherent hierarchical structure (only)

Reason Direct representation of ideas in code eases comprehension and maintenance. Make sure the idea represented in the base class exactly matches all derived types and there is not a better way to express it than using the tight coupling of inheritance.

Do *not* use inheritance when simply having a data member will do. Usually this means that the derived type needs to override a base virtual function or needs access to a protected member.

Example

```
??? Good old Shape example?
```

Example, bad Do *not* represent non-hierarchical domain concepts as class hierarchies.

```
template<typename T>
class Container {
public:
```

```
    // list operations:
    virtual T& get() = 0;
    virtual void put(T&) = 0;
    virtual void insert(Position) = 0;
    // ...
    // vector operations:
    virtual T& operator[](int) = 0;
    virtual void sort() = 0;
    // ...
    // tree operations:
    virtual void balance() = 0;
    // ...
};
```

Here most overriding classes cannot implement most of the functions required in the interface well. Thus the base class becomes an implementation burden. Furthermore, the user of `Container` cannot rely on the member functions actually performing a meaningful operations reasonably efficiently; it may throw an exception instead. Thus users have to resort to run-time checking and/or not using this (over)general interface in favor of a particular interface found by a run-time type inquiry (e.g., a `dynamic_cast`).

Enforcement

- Look for classes with lots of members that do nothing but throw.
- Flag every use of a nonpublic base class `B` where the derived class `D` does not override a virtual function or access a protected member in `B`, and `B` is not one of the following: empty, a template parameter or parameter pack of `D`, a class template specialized with `D`.

C.121: If a base class is used as an interface, make it a pure abstract class

Reason A class is more stable (less brittle) if it does not contain data. Interfaces should normally be composed entirely of public pure virtual functions and a default/empty virtual destructor.

Example

```
class my_interface {
public:
    // ...only pure virtual functions here ...
    virtual ~my_interface() {}   // or =default
};
```

Example, bad

```
class Goof {
public:
    // ...only pure virtual functions here ...
    // no virtual destructor
};

class Derived : public Goof {
    string s;
    // ...
};

void use()
{
    unique_ptr<Goof> p {new Derived{"here we go"}};
    f(p.get()); // use Derived through the Goof interface
    g(p.get()); // use Derived through the Goof interface
 } // leak
```

The `Derived` is `delete`d through its `Goof` interface, so its `string` is leaked. Give `Goof` a virtual destructor and all is well.

Enforcement

- Warn on any class that contains data members and also has an overridable (non-`final`) virtual function.

C.122: Use abstract classes as interfaces when complete separation of interface and implementation is needed

Reason Such as on an ABI (link) boundary.

Example

```
struct Device {
    virtual void write(span<const char> outbuf) = 0;
    virtual void read(span<char> inbuf) = 0;
};

class D1 : public Device {
    // ... data ...
```

```
    void write(span<const char> outbuf) override;
    void read(span<char> inbuf) override;
};

class D2 : public Device {
    // ... differnt data ...

    void write(span<const char> outbuf) override;
    void read(span<char> inbuf) override;
};
```

A user can now use `D1`s and `D2`s interrchangeably through the interface provided by `Device`. Furthermore, we can update `D1` and `D2` in a ways that are not binarily compatible with older versions as long as all access goes through `Device`.

Enforcement

???

C.hierclass: Designing classes in a hierarchy:

C.126: An abstract class typically doesn't need a constructor

Reason An abstract class typically does not have any data for a constructor to initialize.

Example

???

Exceptions

- A base class constructor that does work, such as registering an object somewhere, may need a constructor.
- In extremely rare cases, you might find it reasonable for an abstract class to have a bit of data shared by all derived classes (e.g., use statistics data, debug information, etc.); such classes tend to have constructors. But be warned: Such classes also tend to be prone to requiring virtual inheritance.

Enforcement Flag abstract classes with constructors.

C.127: A class with a virtual function should have a virtual or protected destructor

Reason A class with a virtual function is usually (and in general) used via a pointer to base. Usually, the last user has to call delete on a pointer to base, often via a smart pointer to base, so the destructor should be public and virtual. Less commonly, if deletion through a pointer to base is not intended to be supported, the destructor should be protected and nonvirtual; see C.35.

Example, bad

```
struct B {
    virtual int f() = 0;
    // ... no user-written destructor, defaults to public nonvirtual ...
};

struct D : B {   // bad: class with a resource derived from a class without a virtual
    string s {"default"};
};

void use()
{
    auto p = make_unique<D>();
    // ...
} // calls B::~B only, leaks the string
```

Note There are people who don't follow this rule because they plan to use a class only through a `shared_ptr`: `std::shared_ptr<B> p = std::make_shared<D>(args);` Here, the shared pointer will take care of deletion, so no leak will occur from an inappropriate `delete` of the base. People who do this consistently can get a false positive, but the rule is important – what if one was allocated using `make_unique`? It's not safe unless the author of `B` ensures that it can never be misused, such as by making all constructors private and providing a factory function to enforce the allocation with `make_shared`.

Enforcement

- A class with any virtual functions should have a destructor that is either public and virtual or else protected and nonvirtual.
- Flag `delete` of a class with a virtual function but no virtual destructor.

C.128: Virtual functions should specify exactly one of `virtual`, `override`, or `final`

Reason Readability. Detection of mistakes. Writing explicit `virtual`, `override`, or `final` is self-documenting and enables the compiler to catch mismatch of types and/or names between base and derived classes. However, writing more than one of these three is both redundant and a potential source of errors.

Use `virtual` only when declaring a new virtual function. Use `override` only when declaring an overrider. Use `final` only when declaring an final overrider.

Example, bad

```
struct B {
    void f1(int);
    virtual void f2(int) const;
    virtual void f3(int);
    // ...
};

struct D : B {
    void f1(int);      // warn: D::f1() hides B::f1()
    void f2(int) const;      // warn: no explicit override
    void f3(double);   // warn: D::f3() hides B::f3()
    // ...
};

struct D2 : B {
    virtual void f2(int) final;  // BAD; pitfall, D2::f does not override B::f
};
```

Enforcement

- Compare names in base and derived classes and flag uses of the same name that does not override.
- Flag overrides with neither `override` nor `final`.
- Flag function declarations that use more than one of `virtual`, `override`, and `final`.

C.129: When designing a class hierarchy, distinguish between implementation inheritance and interface inheritance

Reason ??? Herb: I've become a non-fan of implementation inheritance – seems most often an anti-pattern. Are there reasonable examples of it?

Example

```
???
```

Enforcement ???

C.130: Redefine or prohibit copying for a base class; prefer a virtual `clone` function instead

Reason Copying a base is usually slicing. If you really need copy semantics, copy deeply: Provide a virtual `clone` function that will copy the actual most-derived type and return an owning pointer to the new object, and then in derived classes return the derived type (use a covariant return type).

Example

```
class base {
public:
    virtual owner<base*> clone() = 0;
    virtual ~base() = 0;

    base(const base&) = delete;
    base& operator=(const base&) = delete;
};

class derived : public base {
public:
    owner<derived*> clone() override;
    virtual ~derived() override;
};
```

Note that because of language rules, the covariant return type cannot be a smart pointer. See also C.67.

Enforcement

- Flag a class with a virtual function and a non-user-defined copy operation.
- Flag an assignment of base class objects (objects of a class from which another has been derived).

C.131: Avoid trivial getters and setters

Reason A trivial getter or setter adds no semantic value; the data item could just as well be `public`.

Example

```
class point {
    int x;
    int y;
public:
    point(int xx, int yy) : x{xx}, y{yy} { }
    int get_x() { return x; }
    void set_x(int xx) { x = xx; }
    int get_y() { return y; }
    void set_y(int yy) { y = yy; }
    // no behavioral member functions
};
```

Consider making such a class a `struct` – that is, a behaviorless bunch of variables, all public data and no member functions.

```
struct point {
    int x = 0;
    int y = 0;
};
```

Note A getter or a setter that converts from an internal type to an interface type is not trivial (it provides a form of information hiding).

Enforcement Flag multiple `get` and `set` member functions that simply access a member without additional semantics.

C.132: Don't make a function `virtual` without reason

Reason Redundant `virtual` increases run-time and object-code size. A virtual function can be overridden and is thus open to mistakes in a derived class. A virtual function ensures code replication in a templated hierarchy.

Example, bad

```
template<class T>
class Vector {
public:
    // ...
    virtual int size() const { return sz; }   // bad: what good could a derived class d
private:
```

```
    T* elem;   // the elements
    int sz;    // number of elements
};
```

This kind of "vector" isn't meant to be used as a base class at all.

Enforcement

- Flag a class with virtual functions but no derived classes.
- Flag a class where all member functions are virtual and have implementations.

C.133: Avoid `protected` data

Reason `protected` data is a source of complexity and errors. `protected` data complicated the statement of invariants. `protected` data inherently violates the guidance against putting data in base classes, which usually leads to having to deal virtual inheritance as well.

Example

```
???
```

Note Protected member function can be just fine.

Enforcement Flag classes with `protected` data.

C.134: Ensure all non-`const` data members have the same access level

Reason Prevention of logical confusion leading to errors. If the non-`const` data members don't have the same access level, the type is confused about what it's trying to do. Is it a type that maintains an invariant or simply a collection of values?

Discussion The core question is: What code is responsible for maintaining a meaningful/correct value for that variable?

There are exactly two kinds of data members:

- A: Ones that don't participate in the object's invariant. Any combination of values for these members is valid.
- B: Ones that do participate in the object's invariant. Not every combination of values is meaningful (else there'd be no invariant). Therefore all code that has write access to these variables must know about the invariant, know the semantics, and know (and actively implement and enforce) the rules for keeping the values correct.

Data members in category A should just be `public` (or, more rarely, `protected` if you only want derived classes to see them). They don't need encapsulation. All code in the system might as well see and manipulate them.

Data members in category B should be `private` or `const`. This is because encapsulation is important. To make them non-`private` and non-`const` would mean that the object can't control its own state: An unbounded amount of code beyond the class would need to know about the invariant and participate in maintaining it accurately – if these data members were `public`, that would be all calling code that uses the object; if they were `protected`, it would be all the code in current and future derived classes. This leads to brittle and tightly coupled code that quickly becomes a nightmare to maintain. Any code that inadvertently sets the data members to an invalid or unexpected combination of values would corrupt the object and all subsequent uses of the object.

Most classes are either all A or all B:

- *All public*: If you're writing an aggregate bundle-of-variables without an invariant across those variables, then all the variables should be `public`. By convention, declare such classes `struct` rather than `class`
- *All private*: If you're writing a type that maintains an invariant, then all the non-`const` variables should be private – it should be encapsulated.

Exceptions Occasionally classes will mix A and B, usually for debug reasons. An encapsulated object may contain something like non-`const` debug instrumentation that isn't part of the invariant and so falls into category A – it isn't really part of the object's value or meaningful observable state either. In that case, the A parts should be treated as A's (made `public`, or in rarer cases `protected` if they should be visible only to derived classes) and the B parts should still be treated like B's (`private` or `const`).

Enforcement Flag any class that has non-`const` data members with different access levels.

C.135: Use multiple inheritance to represent multiple distinct interfaces

Reason Not all classes will necessarily support all interfaces, and not all callers will necessarily want to deal with all operations. Especially to break apart monolithic interfaces into "aspects" of behavior supported by a given derived class.

Example

```
???
```

Note This is a very common use of inheritance because the need for multiple different interfaces to an implementation is common and such interfaces are often not easily or naturally organized into a single-rooted hierarchy.

Note Such interfaces are typically abstract classes.

Enforcement ???

C.136: Use multiple inheritance to represent the union of implementation attributes

Reason ??? Herb: Here's the second mention of implementation inheritance. I'm very skeptical, even of single implementation inheritance, never mind multiple implementation inheritance which just seems frightening – I don't think that even policy-based design really needs to inherit from the policy types. Am I missing some good examples, or could we consider discouraging this as an anti-pattern?

Example

```
???
```

Note This a relatively rare use because implementation can often be organized into a single-rooted hierarchy.

Enforcement ??? Herb: How about opposite enforcement: Flag any type that inherits from more than one non-empty base class?

C.137: Use `virtual` bases to avoid overly general base classes

Reason ???

Example

```
???
```

Note ???

Enforcement ???

C.138: Create an overload set for a derived class and its bases with `using`

Reason ???

Example

???

C.139: Use `final` sparingly

Reason Capping a hierarchy with `final` is rarely needed for logical reasons and can be damaging to the extensibility of a hierarchy. Capping an individual virtual function with `final` is error-prone as that `final` can easily be overlooked when defining/overriding a set of functions.

Example, bad

```
class Widget { /* ... */ };

class My_widget final : public Widget { /* ... */ };    // nobody will ever want to imp

class My_improved_widget : public My_widget { /* ... */ };  // error: can't do that
```

Example, bad

```
struct Interface {
    virtual int f() = 0;
    virtual int g() = 0;
};

class My_implementation : public Interface {
    int f() override;
    int g() final;  // I want g() to be FAST!
    // ...
};

class Better_implementation : public My_implementation {
    int f();
    int g();
    // ...
};

void use(Interface* p)
{
    int x = p->f();    // Better_implementation::f()
    int y = p->g();    // My_implementation::g() Surprise?
}
```

```
// ...

use(new Better_interface{});
```

The problem is easy to see in a small example, but in a large hierarchy with many virtual functions, tools are required for reliably spotting such problems. Consistent use of `override` would catch this.

Note Claims of performance improvements from `final` should be substantiated. Too often, such claims are based on conjecture or experience with other languages.

There are examples where `final` can be important for both logical and performance reasons. One example is a performance-critical AST hierarchy in a compiler or language analysis tool. New derived classes are not added every year and only by library implementers. However, misuses are (or at least has been) far more common.

Enforcement Flag uses of `final`.

C.140: Do not provide different default arguments for a virtual function and an overrider

Reason That can cause confusion: An overrider do not inherit default arguments..

Example, bad

```
class base {
public:
    virtual int multiply(int value, int factor = 2) = 0;
};

class derived : public base {
public:
    int multiply(int value, int factor = 10) override;
};

derived d;
base& b = d;

b.multiply(10);  // these two calls will call the same function but
d.multiply(10);  // with different arguments and so different results
```

Enforcement Flag default arguments on virtual functions if they differ between base and derived declarations.

C.hier-access: Accessing objects in a hierarchy

C.145: Access polymorphic objects through pointers and references

Reason If you have a class with a virtual function, you don't (in general) know which class provided the function to be used.

Example

```
struct B { int a; virtual int f(); };
struct D : B { int b; int f() override; };

void use(B b)
{
    D d;
    B b2 = d;   // slice
    B b3 = b;
}

void use2()
{
    D d;
    use(d);   // slice
}
```

Both `d`s are sliced.

Exception You can safely access a named polymorphic object in the scope of its definition, just don't slice it.

```
void use3()
{
    D d;
    d.f();   // OK
}
```

Enforcement Flag all slicing.

C.146: Use `dynamic_cast` where class hierarchy navigation is unavoidable

Reason `dynamic_cast` is checked at run time.

Example

```
struct B {   // an interface
    virtual void f();
    virtual void g();
};

struct D : B {   // a wider interface
    void f() override;
    virtual void h();
};

void user(B* pb)
{
    if (D* pd = dynamic_cast<D*>(pb)) {
        // ... use D's interface ...
    }
    else {
        // ... make do with B's interface ...
    }
}
```

Note Like other casts, `dynamic_cast` is overused. Prefer virtual functions to casting. Prefer static polymorphism to hierarchy navigation where it is possible (no run-time resolution necessary) and reasonably convenient.

Note Some people use `dynamic_cast` where a `typeid` would have been more appropriate; `dynamic_cast` is a general "is kind of" operation for discovering the best interface to an object, whereas `typeid` is a "give me the exact type of this object" operation to discover the actual type of an object. The latter is an inherently simpler operation that ought to be faster. The latter (`typeid`) is easily hand-crafted if necessary (e.g., if working on a system where RTTI is - for some reason - prohibited), the former (`dynamic_cast`) is far harder to implement correctly in general.

Consider:

```
struct B {
    const char * name {"B"};
    virtual const char* id() const { return name; }
```

```
    // ...
};

struct D : B {
    const char * name {"D"};
    const char* id() const override { return name; }
    // ...
};

void use()
{
    B* pb1 = new B;
    B* pb2 = new D;

    cout << pb1->id(); // "B"
    cout << pb2->id(); // "D"

    if (pb1->id() == pb2->id()) // *pb1 is the same type as *pb2
    if (pb2 == "D") {         // looks innocent
            D* pd = static_cast<D*>(pb1);
            // ...
    }
    // ...
}
```

The result of `pb2 == "D"` is actually implementation defined. We added it to warn of the dangers of home-brew RTTI. This code may work as expected for years, just to fail on a new machine, new compiler, or a new linker that does not unify character literals.

If you implement your own RTTI, be careful.

Exceptions If your implementation provided a really slow `dynamic_cast`, you may have to use a workaround. However, all workarounds that cannot be statically resolved involve explicit casting (typically `static_cast`) and are error-prone. You will basically be crafting your own special-purpose `dynamic_cast`. So, first make sure that your `dynamic_cast` really is as slow as you think it is (there are a fair number of unsupported rumors about) and that your use of `dynamic_cast` is really performance critical.

We are of the opinion that current implementations of `dynamic_cast` are unnecessarily slow. For example, under suitable conditions, it is possible to perform a `dynamic_cast` in fast constant time. However, compatibility makes changes difficult even if all agree that an effort to optimize is worthwhile.

In very rare cases, if you have measured that the `dynamic_cast` overhead is material, you have other means to statically guarantee that a downcast will succeed (e.g., you are using CRTP carefully), and there is no virtual inheritance involved, consider tactically resorting `static_cast` with a prominent comment and disclaimer summarizing this paragraph and that human attention is needed under maintenance because the type system can't verify correctness. Even so, in our experience such "I know what I'm doing" situations are still a known bug source.

Enforcement Flag all uses of `static_cast` for downcasts, including C-style casts that perform a `static_cast`.

C.147: Use `dynamic_cast` to a reference type when failure to find the required class is considered an error

Reason Casting to a reference expresses that you intend to end up with a valid object, so the cast must succeed. `dynamic_cast` will then throw if it does not succeed.

Example

```
???
```

Enforcement ???

C.148: Use `dynamic_cast` to a pointer type when failure to find the required class is considered a valid alternative

Reason ???

Example

```
???
```

Enforcement ???

C.149: Use `unique_ptr` or `shared_ptr` to avoid forgetting to `delete` objects created using `new`

Reason Avoid resource leaks.

Example

```
void use(int i)
{
    auto p = new int {7};           // bad: initialize local pointers with new
    auto q = make_unique<int>(9);   // ok: guarantee the release of the memory allocate
    if (0 < i) return;              // maybe return and leak
    delete p;                       // too late
}
```

Enforcement

- Flag initialization of a naked pointer with the result of a `new`
- Flag `delete` of local variable

C.150: Use `make_unique()` to construct objects owned by `unique_ptr`s

Reason `make_unique` gives a more concise statement of the construction. It also ensures exception safety in complex expressions.

Example

```
unique_ptr<Foo> p {new<Foo>{7}};   // OK: but repetitive

auto q = make_unique<Foo>(7);      // Better: no repetition of Foo

// Not exception-safe: the compiler may interleave the computations of arguments as fo
//
// 1. allocate memory for Foo,
// 2. construct Foo,
// 3. call bar,
// 4. construct unique_ptr<Foo>.
//
// If bar throws, Foo will not be destroyed, and the memory allocated for it will leak
f(unique_ptr<Foo>(new Foo()), bar());

// Exception-safe: calls to functions are never interleaved.
f(make_unique<Foo>(), bar());
```

Enforcement

- Flag the repetitive usage of template specialization list `<Foo>`
- Flag variables declared to be `unique_ptr<Foo>`

C.151: Use `make_shared()` to construct objects owned by `shared_ptr`s

Reason `make_shared` gives a more concise statement of the construction. It also gives an opportunity to eliminate a separate allocation for the reference counts, by placing the `shared_ptr`'s use counts next to its object.

Example

```
shared_ptr<Foo> p {new<Foo>{7}};    // OK: but repetitive; and separate allocations for

auto q = make_shared<Foo>(7);   // Better: no repetition of Foo; one object
```

Enforcement

- Flag the repetitive usage of template specialization list`<Foo>`
- Flag variables declared to be `shared_ptr<Foo>`

C.152: Never assign a pointer to an array of derived class objects to a pointer to its base

Reason Subscripting the resulting base pointer will lead to invalid object access and probably to memory corruption.

Example

```
struct B { int x; };
struct D : B { int y; };

void use(B*);

D a[] = { {1, 2}, {3, 4}, {5, 6} };
B* p = a;     // bad: a decays to &a[0] which is converted to a B*
p[1].x = 7;   // overwrite D[0].y

use(a);       // bad: a decays to &a[0] which is converted to a B*
```

Enforcement

- Flag all combinations of array decay and base to derived conversions.
- Pass an array as a `span` rather than as a pointer, and don't let the array name suffer a derived-to-base conversion before getting into the `span`

C.over: Overloading and overloaded operators

You can overload ordinary functions, template functions, and operators. You cannot overload function objects.

Overload rule summary:

- C.160: Define operators primarily to mimic conventional usage
- C.161: Use nonmember functions for symmetric operators
- C.162: Overload operations that are roughly equivalent
- C.163: Overload only for operations that are roughly equivalent
- C.164: Avoid conversion operators
- C.165: Use `using` for customization points
- C.166: Overload unary `&` only as part of a system of smart pointers and references
- C.167: Use an operator for an operation with its conventional meaning
- C.168: Define overloaded operators in the namespace of their operands
- C.170: If you feel like overloading a lambda, use a generic lambda

C.160: Define operators primarily to mimic conventional usage

Reason Minimize surprises.

Example

```
class X {
public:
    // ...
    X& operator=(const X&); // member function defining assignment
    friend bool operator==(const X&, const X&); // == needs access to representation
                                                // after a=b we have a==b
    // ...
};
```

Here, the conventional semantics is maintained: Copies compare equal.

Example, bad

```
X operator+(X a, X b) { return a.v - b.v; }   // bad: makes + subtract
```

Note Non-member operators should be either friends or defined in the same namespace as their operands. Binary operators should treat their operands equivalently.

Enforcement Possibly impossible.

C.161: Use nonmember functions for symmetric operators

Reason If you use member functions, you need two. Unless you use a non-member function for (say) `==`, `a == b` and `b == a` will be subtly different.

Example

```
bool operator==(Point a, Point b) { return a.x == b.x && a.y == b.y; }
```

Enforcement Flag member operator functions.

C.162: Overload operations that are roughly equivalent

Reason Having different names for logically equivalent operations on different argument types is confusing, leads to encoding type information in function names, and inhibits generic programming.

Example Consider:

```
void print(int a);
void print(int a, int base);
void print(const string&);
```

These three functions all print their arguments (appropriately). Conversely:

```
void print_int(int a);
void print_based(int a, int base);
void print_string(const string&);
```

These three functions all print their arguments (appropriately). Adding to the name just introduced verbosity and inhibits generic code.

Enforcement ???

C.163: Overload only for operations that are roughly equivalent

Reason Having the same name for logically different functions is confusing and leads to errors when using generic programming.

Example Consider:

```
void open_gate(Gate& g);   // remove obstacle from garage exit lane
void fopen(const char*name, const char* mode);   // open file
```

The two operations are fundamentally different (and unrelated) so it is good that their names differ. Conversely:

```
void open(Gate& g);   // remove obstacle from garage exit lane
void open(const char*name, const char* mode ="r");   // open file
```

The two operations are still fundamentally different (and unrelated) but the names have been reduced to their (common) minimum, opening opportunities for confusion. Fortunately, the type system will catch many such mistakes.

Note Be particularly careful about common and popular names, such as `open`, `move`, `+`, and `==`.

Enforcement ???

C.164: Avoid conversion operators

Reason Implicit conversions can be essential (e.g., `double` to `int`) but often cause surprises (e.g., `String` to C-style string).

Note Prefer explicitly named conversions until a serious need is demonstrated. By "serious need" we mean a reason that is fundamental in the application domain (such as an integer to complex number conversion) and frequently needed. Do not introduce implicit conversions (through conversion operators or non-`explicit` constructors) just to gain a minor convenience.

Example, bad

```
class String {   // handle ownership and access to a sequence of characters
    // ...
    String(czstring p); // copy from *p to *(this->elem)
    // ...
    operator zstring() { return elem; }
    // ...
};
```

```
void user(zstring p)
{
    if (*p == "") {
        String s {"Trouble ahead!"};
        // ...
        p = s;
    }
    // use p
}
```

The string allocated for `s` and assigned to `p` is destroyed before it can be used.

Enforcement Flag all conversion operators.

C.165: Use `using` for customization points

Reason To find function objects and functions defined in a separate namespace to "customize" a common function.

Example Consider `swap`. It is a general (standard library) function with a definition that will work for just about any type. However, it is desirable to define specific `swap()`s for specific types. For example, the general `swap()` will copy the elements of two `vector`s being swapped, whereas a good specific implementation will not copy elements at all.

```
namespace N {
    My_type X { /* ... */ };
    void swap(X&, X&);   // optimized swap for N::X
    // ...
}

void f1(N::X& a, N::X& b)
{
    std::swap(a, b);   // probably not what we wanted: calls std::swap()
}
```

The `std::swap()` in `f1()` does exactly what we asked it to do: it calls the `swap()` in namespace `std`. Unfortunately, that's probably not what we wanted. How do we get `N::X` considered?

```
void f2(N::X& a, N::X& b)
{
    swap(a,b);   // calls N::swap
}
```

But that may not be what we wanted for generic code. There, we typically want the specific function if it exists and the general function if not. This is done by including the general function in the lookup for the function:

```
void f3(N::X& a, N::X& b)
{
    using std::swap;  // make std::swap available
    swap(a,b);        // calls N::swap if it exists, otherwise std::swap
}
```

Enforcement Unlikely, except for known customization points, such as `swap`. The problem is that the unqualified and qualified lookups both have uses.

C.166: Overload unary & only as part of a system of smart pointers and references

Reason The `&` operator is fundamental in C++. Many parts of the C++ semantics assumes its default meaning.

Example

```
class Ptr { // a somewhat smart pointer
    Ptr(X* pp) :p(pp) { /* check */ }
    X* operator->() { /* check */ return p; }
    X operator[](int i);
    X operator*();
private:
    T* p;
};

class X {
    Ptr operator&() { return Ptr{this}; }
    // ...
};
```

Note If you "mess with" operator `&` be sure that its definition has matching meanings for `->`, `[]`, `*`, and `.` on the result type. Note that operator `.` currently cannot be overloaded so a perfect system is impossible. We hope to remedy that: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4477.pdf. Note that `std::addressof()` always yields a built-in pointer.

Enforcement Tricky. Warn if `&` is user-defined without also defining `->` for the result type.

C.168: Define overloaded operators in the namespace of their operands

Reason Readability. Ability for find operators using ADL. Avoiding inconsistent definition in different namespaces

Example

```
struct S { };
bool operator==(S,S);   // OK: in the same namespace as S, and even next to S
S s;

bool s==s;
```

This is what a deafault == would do, if we had such defaults.

Example

```
namespace N {
    struct S { };
    bool operator==(S,S);   // OK: in the same namespace as S, and even next to S
}

N::S s;

bool s==s;  // finds N::operator==() by ADL
```

Example, bad

```
struct S { };
S s;

namespace N {
    S::operator!(S a) { return true; }
    S not_s = !s;
}

namespace M {
    S::operator!(S a) { return false; }
    S not_s = !s;
}
```

Here, the meaning of `!s` differs in `N` and `M`. This can be most confusing. Remove the definition of `namespace M` and the confusion is replaced by an opportunity to make the mistake.

Note If a binary operator is defined for two types that are defined in different namespaces, you cannot follow this rule. For example:

```
Vec::Vector operator\*(const Vec::Vector&, const Mat::Matrix&);
```

This may be something best avoided.

See also This is a special case of the rule that helper functions should be defined in the same namespace as their class.

Enforcement

- Flag operator definitions that are not it the namespace of their operands

C.167: Use an operator for an operation with its conventional meaning

Reason Readability. Convention. Reusability. Support for generic code

Example

```
void cout_my_class(const my_class& c) // confusing, not conventional,not generic
{
    std::cout << /* class members here */;
}

std::ostream& operator<<(std::ostream& os, const my_class& c) //OK
{
    return os << /* class members here */;
}
```

By itself, `cout_my_class` would be OK, but it is not usable/composabe with code that rely on the `<<` convention for output:

```
My_class var { /* ... */ };
// ...
cout << "var = " << var << '\n';
```

Note There are strong and vigorous conventions for the meaning most operators, such as

- comparisons (`==`, `!=`, `<`, `<=`, `>`, and `>=`),
- arithmetic operations (`+`, `-`, `*`, `/`, and `%`)
- access operations (`.`, `->`, unary `*`, and `[]`)
- assignment (`=`)

Don't define those unconventionally and don't invent your own names for them.

Enforcement Tricky. Requires semantic insight.

C.170: If you feel like overloading a lambda, use a generic lambda

Reason You cannot overload by defining two different lambdas with the same name.

Example

```
void f(int);
void f(double);
auto f = [](char);   // error: cannot overload variable and function

auto g = [](int) { /* ... */ };
auto g = [](double) { /* ... */ };   // error: cannot overload variables

auto h = [](auto) { /* ... */ };   // OK
```

Enforcement The compiler catches the attempt to overload a lambda.

C.union: Unions

???

Union rule summary:

- C.180: Use `unions` to ???
- C.181: Avoid "naked" `unions`
- C.182: Use anonymous `unions` to implement tagged unions
- ???

C.180: Use `unions` to ???

??? When should unions be used, if at all? What's a good future-proof way to re-interpret object representations of PODs? ??? variant

Reason ???

Example

```
???
```

Enforcement ???

C.181: Avoid "naked" `union`s

Reason Naked unions are a source of type errors.

Alternative: Wrap them in a class together with a type field.

Alternative: Use `variant`.

Example

```
???
```

Enforcement ???

C.182: Use anonymous `union`s to implement tagged unions

Reason ???

Example

```
???
```

Enforcement ???

Enum: Enumerations

Enumerations are used to define sets of integer values and for defining types for such sets of values. There are two kind of enumerations, "plain" `enum`s and `class enum`s.

Enumeration rule summary:

- Enum.1: Prefer enums over macros
- Enum.2: Use enumerations to represent sets of named constants
- Enum.3: Prefer class enums over "plain" enums
- Enum.4: Define operations on enumerations for safe and simple use
- Enum.5: Don't use `ALL_CAPS` for enumerators
- Enum.6: Use unnamed enumerations for ???
- ???

Enum.1: Prefer enums over macros

Reason Macros do not obey scope and type rules. Also, macro names are removed during preprocessing and so usually don't appear in tools like debuggers.

Example First some bad old code:

```
// webcolors.h (third party header)
#define RED   0xFF0000
#define GREEN 0x00FF00
#define BLUE  0x0000FF

// productinfo.h
// The following define product subtypes based on color
#define RED    0
#define PURPLE 1
#define BLUE   2

int webby = BLUE;   // webby == 2; probably not what was desired
```

Instead use an `enum`:

```
enum class Webcolor { red = 0xFF0000, green = 0x00FF00, blue = 0x0000FF };
enum class Productinfo { red = 0, purple = 1, blue = 2 };

int webby = blue;   // error: be specific
Webcolor webby = Webcolor::blue;
```

Enforcement Flag macros that define integer values

Enum.2: Use enumerations to represent sets of named constants

Reason An enumeration shows the enumerators to be related and can be a named type

Example

```
enum class Webcolor { red = 0xFF0000, green = 0x00FF00, blue = 0x0000FF };
```

Enforcement ???

Enum.3: Prefer class enums over "plain" enums

Reason To minimize surprises: traditional enums convert to int too readily.

Example

```
void PrintColor(int color);

enum Webcolor { red = 0xFF0000, green = 0x00FF00, blue = 0x0000FF };
enum Productinfo { Red=0, Purple=1, Blue=2 };

Webcolor webby = Webcolor::blue;

// Clearly at least one of these calls is buggy.
PrintColor(webby);
PrintColor(Productinfo::Blue);
```

Instead use an `enum class`:

```
void PrintColor(int color);

enum class Webcolor { red=0xFF0000, green=0x00FF00, blue=0x0000FF };
enum class Productinfo { red=0, purple=1, blue=2 };

Webcolor webby = Webcolor::blue;
PrintColor(webby);  // Error: cannot convert Webcolor to int.
PrintColor(Productinfo::Red);  // Error: cannot convert Productinfo to int.
```

Enforcement (Simple) Warn on any non-class enum definition.

Enum.4: Define operations on enumerations for safe and simple use

Reason Convenience of use and avoidance of errors.

Example

???

Enforcement ???

Enum.5: Don't use `ALL_CAPS` for enumerators

Reason Avoid clashes with macros.

Example

```
???
```

Enforcement ???

Enum.6: Use unnamed enumerations for ???

Reason ???

Example

```
???
```

Enforcement ???

R: Resource management

This section contains rules related to resources. A resource is anything that must be acquired and (explicitly or implicitly) released, such as memory, file handles, sockets, and locks. The reason it must be released is typically that it can be in short supply, so even delayed release may do harm. The fundamental aim is to ensure that we don't leak any resources and that we don't hold a resource longer than we need to. An entity that is responsible for releasing a resource is called an owner.

There are a few cases where leaks can be acceptable or even optimal: If you are writing a program that simply produces an output based on an input and the amount of memory needed is proportional to the size of the input, the optimal strategy (for performance and ease of programming) is sometimes simply never to delete anything. If you have enough memory to handle your largest input, leak away, but be sure to give a good error message if you are wrong. Here, we ignore such cases.

- Resource management rule summary:
- R.1: Manage resources automatically using resource handles and RAII (Resource Acquisition Is Initialization)
- R.2: In interfaces, use raw pointers to denote individual objects (only)

- R.3: A raw pointer (a `T*`) is non-owning
- R.4: A raw reference (a `T&`) is non-owning
- R.5: Prefer scoped objects
- R.6: Avoid non-`const` global variables
- Allocation and deallocation rule summary:
- R.10: Avoid `malloc()` and `free()`
- R.11: Avoid calling `new` and `delete` explicitly
- R.12: Immediately give the result of an explicit resource allocation to a manager object
- R.13: Perform at most one explicit resource allocation in a single expression statement
- R.14: ??? array vs. pointer parameter
- R.15: Always overload matched allocation/deallocation pairs
- Smart pointer rule summary:
- R.20: Use `unique_ptr` or `shared_ptr` to represent ownership
- R.21: Prefer `unique_ptr` over `shared_ptr` unless you need to share ownership
- R.22: Use `make_shared()` to make `shared_ptr`s
- R.23: Use `make_unique()` to make `unique_ptr`s
- R.24: Use `std::weak_ptr` to break cycles of `shared_ptr`s
- R.30: Take smart pointers as parameters only to explicitly express lifetime semantics
- R.31: If you have non-`std` smart pointers, follow the basic pattern from `std`
- R.32: Take a `unique_ptr<widget>` parameter to express that a function assumes ownership of a `widget`
- R.33: Take a `unique_ptr<widget>&` parameter to express that a function reseats the`widget`
- R.34: Take a `shared_ptr<widget>` parameter to express that a function is part owner
- R.35: Take a `shared_ptr<widget>&` parameter to express that a function might reseat the shared pointer
- R.36: Take a `const shared_ptr<widget>&` parameter to express that it might retain a reference count to the object ???
- R.37: Do not pass a pointer or reference obtained from an aliased smart pointer

R.1: Manage resources automatically using resource handles and RAII (Resource Acquisition Is Initialization)

Reason To avoid leaks and the complexity of manual resource management. C++'s language-enforced constructor/destructor symmetry mirrors the symmetry inherent in resource acquire/release function pairs such as `fopen`/`fclose`, `lock`/`unlock`, and `new`/`delete`. Whenever you deal with a resource that needs paired acquire/release function calls, encapsulate that resource in an object that enforces pairing for you – acquire the resource in its constructor, and release it in its destructor.

Example, bad Consider:

```
void send(X* x, cstring_span destination)
{
    auto port = OpenPort(destination);
    my_mutex.lock();
    // ...
    Send(port, x);
    // ...
    my_mutex.unlock();
    ClosePort(port);
    delete x;
}
```

In this code, you have to remember to `unlock`, `ClosePort`, and `delete` on all paths, and do each exactly once. Further, if any of the code marked `...` throws an exception, then `x` is leaked and `my_mutex` remains locked.

Example Consider:

```
void send(unique_ptr<X> x, cstring_span destination)  // x owns the X
{
    Port port{destination};            // port owns the PortHandle
    lock_guard<mutex> guard{my_mutex}; // guard owns the lock
    // ...
    Send(port, x);
    // ...
} // automatically unlocks my_mutex and deletes the pointer in x
```

Now all resource cleanup is automatic, performed once on all paths whether or not there is an exception. As a bonus, the function now advertises that it takes over ownership of the pointer.

What is `Port`? A handy wrapper that encapsulates the resource:

```
class Port {
    PortHandle port;
public:
    Port(cstring_span destination) : port{OpenPort(destination)} { }
    ~Port() { ClosePort(port); }
    operator PortHandle() { return port; }

    // port handles can't usually be cloned, so disable copying and assignment if nece
    Port(const Port&) =delete;
    Port& operator=(const Port&) =delete;
};
```

Note Where a resource is "ill-behaved" in that it isn't represented as a class with a destructor, wrap it in a class or use `finally`

See also: RAII.

R.2: In interfaces, use raw pointers to denote individual objects (only)

Reason Arrays are best represented by a container type (e.g., `vector` (owning)) or a `span` (non-owning). Such containers and views hold sufficient information to do range checking.

Example, bad

```
void f(int* p, int n)   // n is the number of elements in p[]
{
    // ...
    p[2] = 7;   // bad: subscript raw pointer
    // ...
}
```

The compiler does not read comments, and without reading other code you do not know whether `p` really points to `n` elements. Use a `span` instead.

Example

```
void g(int* p, int fmt)   // print *p using format #fmt
{
    // ... uses *p and p[0] only ...
}
```

Exception: C-style strings are passed as single pointers to a zero-terminated sequence of characters. Use `zstring` rather than `char*` to indicate that you rely on that convention.

Note Many current uses of pointers to a single element could be references. However, where `nullptr` is a possible value, a reference may not be an reasonable alternative.

Enforcement

- Flag pointer arithmetic (including `++`) on a pointer that is not part of a container, view, or iterator. This rule would generate a huge number of false positives if applied to an older code base.
- Flag array names passed as simple pointers

R.3: A raw pointer (a `T*`) is non-owning

Reason There is nothing (in the C++ standard or in most code) to say otherwise and most raw pointers are non-owning. We want owning pointers identified so that we can reliably and efficiently delete the objects pointed to by owning pointers.

Example

```
void f()
{
    int* p1 = new int{7};           // bad: raw owning pointer
    auto p2 = make_unique<int>(7);  // OK: the int is owned by a unique pointer
    // ...
}
```

The `unique_ptr` protects against leaks by guaranteeing the deletion of its object (even in the presence of exceptions). The `T*` does not.

Example

```
template<typename T>
class X {
    // ...
public:
    T* p;   // bad: it is unclear whether p is owning or not
    T* q;   // bad: it is unclear whether q is owning or not
};
```

We can fix that problem by making ownership explicit:

```
template<typename T>
class X2 {
    // ...
public:
    owner<T> p;    // OK: p is owning
    T* q;          // OK: q is not owning
};
```

Exceptions A major class of exception is legacy code, especially code that must remain compilable as C or interface with C and C-style C++ through ABIs. The fact that there are billions of lines of code that violate this rule against owning `T*`s cannot be ignored. We'd love to see program transformation tools turning 20-year-old "legacy" code into shiny modern code, we encourage the development, deployment and use of such tools, we hope the guidelines will help the development of such tools, and we even contributed (and contribute) to the research and development in this area. However, it will take time: "legacy code" is generated faster than we can renovate old code, and so it will be for a few years.

This code cannot all be rewritten (ever assuming good code transformation software), especially not soon. This problem cannot be solved (at scale) by transforming all owning pointers to `unique_ptr`s and `shared_ptr`s, partly because we need/use owning "raw pointers" as well as simple pointers in the implementation of our fundamental resource handles. For example, common `vector` implementations have one owning pointer and two non-owning pointers. Many ABIs (and essentially all interfaces to C code) use `T*`s, some of them owning. Some interfaces cannot be simply annotated with `owner` because they need to remain compilable as C (although this would be a rare good use for a macro, that expands to `owner` in C++ mode only).

Note `owner<T>` has no default semantics beyond `T*`. It can be used without changing any code using it and without affecting ABIs. It is simply a indicator to programmers and analysis tools. For example, if an `owner<T>` is a member of a class, that class better have a destructor that `delete`s it.

Example, bad Returning a (raw) pointer imposes a life-time management uncertainty on the caller; that is, who deletes the pointed-to object?

```
Gadget* make_gadget(int n)
{
    auto p = new Gadget{n};
    // ...
    return p;
}

void caller(int n)
```

```
{
    auto p = make_gadget(n);   // remember to delete p
    // ...
    delete p;
}
```

In addition to suffering from the problem from leak, this adds a spurious allocation and deallocation operation, and is needlessly verbose. If Gadget is cheap to move out of a function (i.e., is small or has an efficient move operation), just return it "by value" (see "out" return values):

```
Gadget make_gadget(int n)
{
    Gadget g{n};
    // ...
    return g;
}
```

Note This rule applies to factory functions.

Note If pointer semantics are required (e.g., because the return type needs to refer to a base class of a class hierarchy (an interface)), return a "smart pointer."

Enforcement

- (Simple) Warn on `delete` of a raw pointer that is not an `owner<T>`.
- (Moderate) Warn on failure to either `reset` or explicitly `delete` an `owner<T>` pointer on every code path.
- (Simple) Warn if the return value of `new` or a function call with return value of pointer type is assigned to a raw pointer.
- (Simple) Warn if a function returns an object that was allocated within the function but has a move constructor. Suggest considering returning it by value instead.

R.4: A raw reference (a `T&`) is non-owning

Reason There is nothing (in the C++ standard or in most code) to say otherwise and most raw references are non-owning. We want owners identified so that we can reliably and efficiently delete the objects pointed to by owning pointers.

Example

```
void f()
{
    int& r = *new int{7};  // bad: raw owning reference
    // ...
    delete &r;             // bad: violated the rule against deleting raw pointers
}
```

See also: The raw pointer rule

Enforcement See the raw pointer rule

R.5: Don't heap-allocate unnecessarily

Reason A scoped object is a local object, a global object, or a member. This implies that there is no separate allocation and deallocation cost in excess of that already used for the containing scope or object. The members of a scoped object are themselves scoped and the scoped object's constructor and destructor manage the members' lifetimes.

Example The following example is inefficient (because it has unnecessary allocation and deallocation), vulnerable to exception throws and returns in the "¦ part (leading to leaks), and verbose:

```
void some_function(int n)
{
    auto p = new Gadget{n};
    // ...
    delete p;
}
```

Instead, use a local variable:

```
void some_function(int n)
{
    Gadget g{n};
    // ...
}
```

Enforcement

- (Moderate) Warn if an object is allocated and then deallocated on all paths within a function. Suggest it should be a local `auto` stack object instead.
- (Simple) Warn if a local `Unique_ptr` or `Shared_ptr` is not moved, copied, reassigned or `reset` before its lifetime ends.

R.6: Avoid non-`const` global variables

Reason Global variables can be accessed from everywhere so they can introduce surprising dependencies between apparently unrelated objects. They are a notable source of errors.

Warning: The initialization of global objects is not totally ordered. If you use a global object initialize it with a constant. Note that it is possible to get undefined initialization order even for `const` objects.

Exception: A global object is often better than a singleton.

Exception: An immutable (`const`) global does not introduce the problems we try to avoid by banning global objects.

Enforcement (??? NM: Obviously we can warn about non-`const` statics ... do we want to?)

R.alloc: Allocation and deallocation

R.10: Avoid `malloc()` and `free()`

Reason `malloc()` and `free()` do not support construction and destruction, and do not mix well with `new` and `delete`.

Example

```
class Record {
    int id;
    string name;
    // ...
};

void use()
{
    Record* p1 = static_cast<Record*>(malloc(sizeof(Record)));
    // p1 may be nullptr
    // *p1 is not initialized; in particular, that string isn't a string, but a string

    auto p2 = new Record;

    // unless an exception is thrown, *p2 is default initialized
    auto p3 = new(nothrow) Record;
    // p3 may be nullptr; if not, *p3 is default initialized

    // ...
```

```
    delete p1;    // error: cannot delete object allocated by malloc()
    free(p2);    // error: cannot free() object allocated by new
}
```

In some implementations that `delete` and that `free()` might work, or maybe they will cause run-time errors.

Exception There are applications and sections of code where exceptions are not acceptable. Some of the best such examples are in life-critical hard real-time code. Beware that many bans on exception use are based on superstition (bad) or by concerns for older code bases with unsystematic resource management (unfortunately, but sometimes necessary). In such cases, consider the `nothrow` versions of `new`.

Enforcement Flag explicit use of `malloc` and `free`.

R.11: Avoid calling `new` and `delete` explicitly

Reason The pointer returned by `new` should belong to a resource handle (that can call `delete`). If the pointer returned by `new` is assigned to a plain/naked pointer, the object can be leaked.

Note In a large program, a naked `delete` (that is a `delete` in application code, rather than part of code devoted to resource management) is a likely bug: if you have N `delete`s, how can you be certain that you don't need N+1 or N-1? The bug may be latent: it may emerge only during maintenance. If you have a naked `new`, you probably need a naked `delete` somewhere, so you probably have a bug.

Enforcement (Simple) Warn on any explicit use of `new` and `delete`. Suggest using `make_unique` instead.

R.12: Immediately give the result of an explicit resource allocation to a manager object

Reason If you don't, an exception or a return may lead to a leak.

Example, bad

```
void f(const string& name)
{
    FILE* f = fopen(name, "r");          // open the file
```

```
    vector<char> buf(1024);
    auto _ = finally([f] { fclose(f); })  // remember to close the file
    // ...
}
```

The allocation of `buf` may fail and leak the file handle.

Example

```
void f(const string& name)
{
    ifstream f{name, "r"};   // open the file
    vector<char> buf(1024);
    // ...
}
```

The use of the file handle (in `ifstream`) is simple, efficient, and safe.

Enforcement

- Flag explicit allocations used to initialize pointers (problem: how many direct resource allocations can we recognize?)

R.13: Perform at most one explicit resource allocation in a single expression statement

Reason If you perform two explicit resource allocations in one statement, you could leak resources because the order of evaluation of many subexpressions, including function arguments, is unspecified.

Example

```
void fun(shared_ptr<Widget> sp1, shared_ptr<Widget> sp2);
```

This `fun` can be called like this:

```
fun(shared_ptr<Widget>(new Widget(a, b)), shared_ptr<Widget>(new Widget(c, d)));   // BA
```

This is exception-unsafe because the compiler may reorder the two expressions building the function's two arguments. In particular, the compiler can interleave execution of the two expressions: Memory allocation (by calling `operator new`) could be done first for both objects, followed by attempts to call the two `Widget` constructors. If one of the constructor calls throws an exception, then the other object's memory will never be released!

This subtle problem has a simple solution: Never perform more than one explicit resource allocation in a single expression statement. For example:

```
shared_ptr<Widget> sp1(new Widget(a, b)); // Better, but messy
fun(sp1, new Widget(c, d));
```

The best solution is to avoid explicit allocation entirely use factory functions that return owning objects:

```
fun(make_shared<Widget>(a, b), make_shared<Widget>(c, d)); // Best
```

Write your own factory wrapper if there is not one already.

Enforcement

- Flag expressions with multiple explicit resource allocations (problem: how many direct resource allocations can we recognize?)

R.14: ??? array vs. pointer parameter

Reason An array decays to a pointer, thereby losing its size, opening the opportunity for range errors.

Example

```
??? what do we recommend: f(int*[]) or f(int**) ???
```

Alternative: Use `span` to preserve size information.

Enforcement Flag `[]` parameters.

R.15: Always overload matched allocation/deallocation pairs

Reason Otherwise you get mismatched operations and chaos.

Example

```
class X {
    // ...
    void* operator new(size_t s);
    void operator delete(void*);
    // ...
};
```

Note If you want memory that cannot be deallocated, `=delete` the deallocation operation. Don't leave it undeclared.

Enforcement Flag incomplete pairs.

R.smart: Smart pointers

R.20: Use `unique_ptr` or `shared_ptr` to represent ownership

Reason They can prevent resource leaks.

Example Consider:

```
void f()
{
    X x;
    X* p1 { new X };              // see also ???
    unique_ptr<T> p2 { new X };   // unique ownership; see also ???
    shared_ptr<T> p3 { new X };   // shared ownership; see also ???
}
```

This will leak the object used to initialize `p1` (only).

Enforcement (Simple) Warn if the return value of `new` or a function call with return value of pointer type is assigned to a raw pointer.

R.21: Prefer `unique_ptr` over `shared_ptr` unless you need to share ownership

Reason A `unique_ptr` is conceptually simpler and more predictable (you know when destruction happens) and faster (you don't implicitly maintain a use count).

Example, bad This needlessly adds and maintains a reference count.

```
void f()
{
    shared_ptr<Base> base = make_shared<Derived>();
    // use base locally, without copying it -- refcount never exceeds 1
} // destroy base
```

Example This is more efficient:

```
void f()
{
    unique_ptr<Base> base = make_unique<Derived>();
    // use base locally
} // destroy base
```

Enforcement (Simple) Warn if a function uses a `Shared_ptr` with an object allocated within the function, but never returns the `Shared_ptr` or passes it to a function requiring a `Shared_ptr&`. Suggest using `unique_ptr` instead.

R.22: Use `make_shared()` to make `shared_ptr`s

Reason If you first make an object and then give it to a `shared_ptr` constructor, you (most likely) do one more allocation (and later deallocation) than if you use `make_shared()` because the reference counts must be allocated separately from the object.

Example Consider:

```
shared_ptr<X> p1 { new X{2} }; // bad
auto p = make_shared<X>(2);    // good
```

The `make_shared()` version mentions `X` only once, so it is usually shorter (as well as faster) than the version with the explicit `new`.

Enforcement (Simple) Warn if a `shared_ptr` is constructed from the result of `new` rather than `make_shared`.

R.23: Use `make_unique()` to make `unique_ptr`s

Reason For convenience and consistency with `shared_ptr`.

Note `make_unique()` is C++14, but widely available (as well as simple to write).

Enforcement (Simple) Warn if a `unique_ptr` is constructed from the result of `new` rather than `make_unique`.

R.24: Use `std::weak_ptr` to break cycles of `shared_ptrs`

Reason `shared_ptr`'s rely on use counting and the use count for a cyclic structure never goes to zero, so we need a mechanism to be able to destroy a cyclic structure.

Example

```
???
```

Note ??? (HS: A lot of people say "to break cycles", while I think "temporary shared ownership" is more to the point.) ???(BS: breaking cycles is what you must do; temporarily sharing ownership is how you do it. You could "temporarily share ownership" simply by using another `stared_ptr`.)

Enforcement ??? probably impossible. If we could statically detect cycles, we wouldn't need `weak_ptr`

R.30: Take smart pointers as parameters only to explicitly express lifetime semantics

Reason Accepting a smart pointer to a `widget` is wrong if the function just needs the `widget` itself. It should be able to accept any `widget` object, not just ones whose lifetimes are managed by a particular kind of smart pointer. A function that does not manipulate lifetime should take raw pointers or references instead.

Example, bad

```
// callee
void f(shared_ptr<widget>& w)
{
    // ...
    use(*w); // only use of w -- the lifetime is not used at all
    // ...
};

// caller
shared_ptr<widget> my_widget = /* ... */;
f(my_widget);

widget stack_widget;
f(stack_widget); // error
```

Example, good

```
// callee
void f(widget& w)
{
    // ...
    use(w);
    // ...
};

// caller
shared_ptr<widget> my_widget = /* ... */;
f(*my_widget);

widget stack_widget;
f(stack_widget); // ok -- now this works
```

Enforcement

- (Simple) Warn if a function takes a parameter of a smart pointer type (that overloads `operator->` or `operator*`) `unique_ptr` or `shared_ptr` and the function only calls any of: `operator*`, `operator->` or `get()`.
- Flag a parameter of a smart pointer type (a type that overloads `operator->` or `operator*`) that is copyable but never copied/moved from in the function body or else movable but never moved from in the function body or by being a by-value parameter, and that is never modified, and that is not passed along to another function that could do so. That means the ownership semantics are not used. Suggest using a `T*` or `T&` instead.

R.31: If you have non-`std` smart pointers, follow the basic pattern from `std`

Reason The rules in the following section also work for other kinds of third-party and custom smart pointers and are very useful for diagnosing common smart pointer errors that cause performance and correctness problems. You want the rules to work on all the smart pointers you use.

Any type (including primary template or specialization) that overloads unary `*` and `->` is considered a smart pointer:

- If it is copyable, it is recognized as a reference-counted `shared_ptr`.
- If it is not copyable, it is recognized as a unique `unique_ptr`.

Example

```
// use Boost's intrusive_ptr
#include <boost/intrusive_ptr.hpp>
void f(boost::intrusive_ptr<widget> p)  // error under rule 'sharedptrparam'
{
    p->foo();
}

// use Microsoft's CComPtr
#include <atlbase.h>
void f(CComPtr<widget> p)               // error under rule 'sharedptrparam'
{
    p->foo();
}
```

Both cases are an error under the `sharedptrparam` guideline: `p` is a `Shared_ptr`, but nothing about its sharedness is used here and passing it by value is a silent pessimization; these functions should accept a smart pointer only if they need to participate in the widget's lifetime management. Otherwise they should accept a `widget*`, if it can be `nullptr`. Otherwise, and ideally, the function should accept a `widget&`. These smart pointers match the `Shared_ptr` concept, so these guideline enforcement rules work on them out of the box and expose this common pessimization.

R.32: Take a `unique_ptr<widget>` parameter to express that a function assumes ownership of a `widget`

Reason Using `unique_ptr` in this way both documents and enforces the function call's ownership transfer.

Example

```
void sink(unique_ptr<widget>); // consumes the widget

void sink(widget*);            // just uses the widget
```

Example, bad

```
void thinko(const unique_ptr<widget>&); // usually not what you want
```

Enforcement

- (Simple) Warn if a function takes a `Unique_ptr<T>` parameter by lvalue reference and does not either assign to it or call `reset()` on it on at least one code path. Suggest taking a `T*` or `T&` instead.
- (Simple) ((Foundation)) Warn if a function takes a `Unique_ptr<T>` parameter by reference to `const`. Suggest taking a `const T*` or `const T&` instead.
- (Simple) ((Foundation)) Warn if a function takes a `Unique_ptr<T>` parameter by rvalue reference. Suggest using pass by value instead.

R.33: Take a `unique_ptr<widget>&` parameter to express that a function reseats the`widget`

Reason Using `unique_ptr` in this way both documents and enforces the function call's reseating semantics.

Note "reseat" means "making a reference or a smart pointer refer to a different object."

Example

```
void reseat(unique_ptr<widget>&); // "will" or "might" reseat pointer
```

Example, bad

```
void thinko(const unique_ptr<widget>&); // usually not what you want
```

Enforcement

- (Simple) Warn if a function takes a `Unique_ptr<T>` parameter by lvalue reference and does not either assign to it or call `reset()` on it on at least one code path. Suggest taking a `T*` or `T&` instead.
- (Simple) ((Foundation)) Warn if a function takes a `Unique_ptr<T>` parameter by reference to `const`. Suggest taking a `const T*` or `const T&` instead.
- (Simple) ((Foundation)) Warn if a function takes a `Unique_ptr<T>` parameter by rvalue reference. Suggest using pass by value instead.

R.34: Take a `shared_ptr<widget>` parameter to express that a function is part owner

Reason This makes the function's ownership sharing explicit.

Example, good

```
void share(shared_ptr<widget>);              // share - "will" retain refcount

void reseat(shared_ptr<widget>&);            // "might" reseat ptr

void may_share(const shared_ptr<widget>&); // "might" retain refcount
```

Enforcement

- (Simple) Warn if a function takes a Shared_ptr<T> parameter by lvalue reference and does not either assign to it or call reset() on it on at least one code path. Suggest taking a T* or T& instead.
- (Simple) ((Foundation)) Warn if a function takes a Shared_ptr<T> by value or by reference to const and does not copy or move it to another Shared_ptr on at least one code path. Suggest taking a T* or T& instead.
- (Simple) ((Foundation)) Warn if a function takes a Shared_ptr<T> by rvalue reference. Suggesting taking it by value instead.

R.35: Take a shared_ptr<widget>& parameter to express that a function might reseat the shared pointer

Reason This makes the function's reseating explicit.

Note "reseat" means "making a reference or a smart pointer refer to a different object."

Example, good

```
void share(shared_ptr<widget>);              // share - "will" retain refcount

void reseat(shared_ptr<widget>&);            // "might" reseat ptr

void may_share(const shared_ptr<widget>&); // "might" retain refcount
```

Enforcement

- (Simple) Warn if a function takes a Shared_ptr<T> parameter by lvalue reference and does not either assign to it or call reset() on it on at least one code path. Suggest taking a T* or T& instead.
- (Simple) ((Foundation)) Warn if a function takes a Shared_ptr<T> by value or by reference to const and does not copy or move it to another Shared_ptr on at least one code path. Suggest taking a T* or T& instead.
- (Simple) ((Foundation)) Warn if a function takes a Shared_ptr<T> by rvalue reference. Suggesting taking it by value instead.

R.36: Take a `const shared_ptr<widget>&` parameter to express that it might retain a reference count to the object ???

Reason This makes the function's ??? explicit.

Example, good

```
void share(shared_ptr<widget>);            // share - "will" retain refcount

void reseat(shared_ptr<widget>&);          // "might" reseat ptr

void may_share(const shared_ptr<widget>&); // "might" retain refcount
```

Enforcement

- (Simple) Warn if a function takes a `Shared_ptr<T>` parameter by lvalue reference and does not either assign to it or call `reset()` on it on at least one code path. Suggest taking a `T*` or `T&` instead.
- (Simple) ((Foundation)) Warn if a function takes a `Shared_ptr<T>` by value or by reference to `const` and does not copy or move it to another `Shared_ptr` on at least one code path. Suggest taking a `T*` or `T&` instead.
- (Simple) ((Foundation)) Warn if a function takes a `Shared_ptr<T>` by rvalue reference. Suggesting taking it by value instead.

R.37: Do not pass a pointer or reference obtained from an aliased smart pointer

Reason Violating this rule is the number one cause of losing reference counts and finding yourself with a dangling pointer. Functions should prefer to pass raw pointers and references down call chains. At the top of the call tree where you obtain the raw pointer or reference from a smart pointer that keeps the object alive. You need to be sure that the smart pointer cannot inadvertently be reset or reassigned from within the call tree below.

Note To do this, sometimes you need to take a local copy of a smart pointer, which firmly keeps the object alive for the duration of the function and the call tree.

Example Consider this code:

```
// global (static or heap), or aliased local ...
shared_ptr<widget> g_p = ...;

void f(widget& w)
{
```

```
    g();
    use(w);  // A
}

void g()
{
    g_p = ...; // oops, if this was the last shared_ptr to that widget, destroys the w
}
```

The following should not pass code review:

```
void my_code()
{
    f(*g_p);     // BAD: passing pointer or reference obtained from a nonlocal smart po
                 //      that could be inadvertently reset somewhere inside f or it ca
    g_p->func(); // BAD: same reason, just passing it as a "this" pointer
}
```

The fix is simple – take a local copy of the pointer to "keep a ref count" for your call tree:

```
void my_code()
{
    auto pin = g_p; // cheap: 1 increment covers this entire function and all the call
    f(*pin);        // GOOD: passing pointer or reference obtained from a local unalias
    pin->func();    // GOOD: same reason
}
```

Enforcement

- (Simple) Warn if a pointer or reference obtained from a smart pointer variable (`Unique_ptr` or `Shared_ptr`) that is nonlocal, or that is local but potentially aliased, is used in a function call. If the smart pointer is a `Shared_ptr` then suggest taking a local copy of the smart pointer and obtain a pointer or reference from that instead.

ES: Expressions and Statements

Expressions and statements are the lowest and most direct way of expressing actions and computation. Declarations in local scopes are statements.

For naming, commenting, and indentation rules, see NL: Naming and layout.

General rules:

- ES.1: Prefer the standard library to other libraries and to “handcrafted code”
- ES.2: Prefer suitable abstractions to direct use of language features

Declaration rules:

- ES.5: Keep scopes small
- ES.6: Declare names in for-statement initializers and conditions to limit scope
- ES.7: Keep common and local names short, and keep uncommon and nonlocal names longer
- ES.8: Avoid similar-looking names
- ES.9: Avoid `ALL_CAPS` names
- ES.10: Declare one name (only) per declaration
- ES.11: Use `auto` to avoid redundant repetition of type names
- ES.20: Always initialize an object
- ES.21: Don’t introduce a variable (or constant) before you need to use it
- ES.22: Don’t declare a variable until you have a value to initialize it with
- ES.23: Prefer the `{}`-initializer syntax
- ES.24: Use a `unique_ptr<T>` to hold pointers in code that may throw
- ES.25: Declare an object `const` or `constexpr` unless you want to modify its value later on
- ES.26: Don’t use a variable for two unrelated purposes
- ES.27: Use `std::array` or `stack_array` for arrays on the stack
- ES.28: Use lambdas for complex initialization, especially of `const` variables
- ES.30: Don’t use macros for program text manipulation
- ES.31: Don’t use macros for constants or “functions”
- ES.32: Use `ALL_CAPS` for all macro names
- ES.33: If you must use macros, give them unique names
- ES.40: Don’t define a (C-style) variadic function

Expression rules:

- ES.40: Avoid complicated expressions
- ES.41: If in doubt about operator precedence, parenthesize
- ES.42: Keep use of pointers simple and straightforward
- ES.43: Avoid expressions with undefined order of evaluation
- ES.44: Don’t depend on order of evaluation of function arguments
- ES.45: Avoid narrowing conversions
- ES.46: Avoid “magic constants”; use symbolic constants
- ES.47: Use `nullptr` rather than `0` or `NULL`
- ES.48: Avoid casts
- ES.49: If you must use a cast, use a named cast
- ES.50: Don’t cast away `const`

- ES.55: Avoid the need for range checking
- ES.56: Avoid `std::move()` in application code
- ES.60: Avoid `new` and `delete` outside resource management functions
- ES.61: delete arrays using `delete[]` and non-arrays using `delete`
- ES.62: Don't compare pointers into different arrays
- ES.63: Don't slice

Statement rules:

- ES.70: Prefer a `switch`-statement to an `if`-statement when there is a choice
- ES.71: Prefer a range-`for`-statement to a `for`-statement when there is a choice
- ES.72: Prefer a `for`-statement to a `while`-statement when there is an obvious loop variable
- ES.73: Prefer a `while`-statement to a `for`-statement when there is no obvious loop variable
- ES.74: Prefer to declare a loop variable in the initializer part of as `for`-statement
- ES.75: Avoid `do`-statements
- ES.76: Avoid `goto`
- ES.77: ??? `continue`
- ES.78: Always end a non-empty `case` with a `break`
- ES.79: ??? `default`
- ES.85: Make empty statements visible
- ES.86: Avoid modifying loop control variables inside the body of raw for-loops

Arithmetic rules:

- ES.100: Don't mix signed and unsigned arithmetic
- ES.101: Use unsigned types for bit manipulation
- ES.102: Use signed types for arithmetic
- ES.103: Don't overflow
- ES.104: Don't underflow
- ES.105: Don't divide by zero

ES.1: Prefer the standard library to other libraries and to "handcrafted code"

Reason Code using a library can be much easier to write than code working directly with language features, much shorter, tend to be of a higher level of abstraction, and the library code is presumably already tested. The ISO C++ standard library is among the most widely known and best tested libraries. It is available as part of all C++ Implementations.

Example

```
auto sum = accumulate(begin(a), end(a), 0.0);   // good
```

a range version of `accumulate` would be even better:

```
auto sum = accumulate(v, 0.0); // better
```

but don't hand-code a well-known algorithm:

```
int max = v.size();   // bad: verbose, purpose unstated
double sum = 0.0;
for (int i = 0; i < max; ++i)
    sum = sum + v[i];
```

Exception: Large parts of the standard library rely on dynamic allocation (free store). These parts, notably the containers but not the algorithms, are unsuitable for some hard-real time and embedded applications. In such cases, consider providing/using similar facilities, e.g., a standard-library-style container implemented using a pool allocator.

Enforcement Not easy. ??? Look for messy loops, nested loops, long functions, absence of function calls, lack of use of non-built-in types. Cyclomatic complexity?

ES.2: Prefer suitable abstractions to direct use of language features

Reason A "suitable abstraction" (e.g., library or class) is closer to the application concepts than the bare language, leads to shorter and clearer code, and is likely to be better tested.

Example

```
vector<string> read1(istream& is)   // good
{
    vector<string> res;
    for (string s; is >> s;)
        res.push_back(s);
    return res;
}
```

The more traditional and lower-level near-equivalent is longer, messier, harder to get right, and most likely slower:

```
char** read2(istream& is, int maxelem, int maxstring, int* nread)   // bad: verbose and
{
    auto res = new char*[maxelem];
    int elemcount = 0;
    while (is && elemcount < maxelem) {
        auto s = new char[maxstring];
        is.read(s, maxstring);
        res[elemcount++] = s;
    }
    nread = elemcount;
    return res;
}
```

Once the checking for overflow and error handling has been added that code gets quite messy, and there is the problem remembering to `delete` the returned pointer and the C-style strings that array contains.

Enforcement Not easy. ??? Look for messy loops, nested loops, long functions, absence of function calls, lack of use of non-built-in types. Cyclomatic complexity?

ES.dcl: Declarations

A declaration is a statement. a declaration introduces a name into a scope and may cause the construction of a named object.

ES.5: Keep scopes small

Reason Readability. Minimize resource retention. Avoid accidental misuse of value.

Alternative formulation: Don't declare a name in an unnecessarily large scope.

Example

```
void use()
{
    int i;    // bad: i is needlessly accessible after loop
    for (i = 0; i < 20; ++i) { /* ... */ }
    // no intended use of i here
    for (int i = 0; i < 20; ++i) { /* ... */ }  // good: i is local to for-loop

    if (auto pc = dynamic_cast<Circle*>(ps)) {  // good: pc is local to if-statement
        // ... deal with Circle ...
    }
```

```
    else {
        // ... handle error ...
    }
}
```

Example, bad

```
void use(const string& name)
{
    string fn = name+".txt";
    ifstream is {fn};
    Record r;
    is >> r;
    // ... 200 lines of code without intended use of fn or is ...
}
```

This function is by most measure too long anyway, but the point is that the resources used by `fn` and the file handle held by `is` are retained for much longer than needed and that unanticipated use of `is` and `fn` could happen later in the function. In this case, it might be a good idea to factor out the read:

```
Record load_record(const string& name)
{
    string fn = name+".txt";
    ifstream is {fn};
    Record r;
    is >> r;
    return r;
}

void use(const string& name)
{
    Record r = load_record(name);
    // ... 200 lines of code ...
}
```

Enforcement

- Flag loop variable declared outside a loop and not used after the loop
- Flag when expensive resources, such as file handles and locks are not used for N-lines (for some suitable N)

ES.6: Declare names in for-statement initializers and conditions to limit scope

Reason Readability. Minimize resource retention.

Example

```
void use()
{
    for (string s; cin >> s;)
        v.push_back(s);

    for (int i = 0; i < 20; ++i) {   // good: i is local to for-loop
        // ...
    }

    if (auto pc = dynamic_cast<Circle*>(ps)) {   // good: pc is local to if-statement
        // ... deal with Circle ...
    }
    else {
        // ... handle error ...
    }
}
```

Enforcement

- Flag loop variables declared before the loop and not used after the loop
- (hard) Flag loop variables declared before the loop and used after the loop for an unrelated purpose.

ES.7: Keep common and local names short, and keep uncommon and nonlocal names longer

Reason Readability. Lowering the chance of clashes between unrelated non-local names.

Example Conventional short, local names increase readability:

```
template<typename T>    // good
void print(ostream& os, const vector<T>& v)
{
    for (int i = 0; i < v.end(); ++i)
        os << v[i] << '\n';
}
```

An index is conventionally called `i` and there is no hint about the meaning of the vector in this generic function, so `v` is as good name as any. Compare

```
template<typename Element_type>   // bad: verbose, hard to read
void print(ostream& target_stream, const vector<Element_type>& current_vector)
{
    for (int current_element_index = 0;
            current_element_index < current_vector.end();
            ++current_element_index
    )
    target_stream << current_vector[i] << '\n';
}
```

Yes, it is a caricature, but we have seen worse.

Example Unconventional and short non-local names obscure code:

```
void use1(const string& s)
{
    // ...
    tt(s);   // bad: what is tt()?
    // ...
}
```

Better, give non-local entities readable names:

```
void use1(const string& s)
{
    // ...
    trim_tail(s);   // better
    // ...
}
```

Here, there is a chance that the reader knows what `trim_tail` means and that the reader can remember it after looking it up.

Example, bad Argument names of large functions are de facto non-local and should be meaningful:

```
void complicated_algorithm(vector<Record>& vr, const vector<int>& vi, map<string, int>&
// read from events in vr (marking used Records) for the indices in vi placing (name,
{
    // ... 500 lines of code using vr, vi, and out ...
}
```

We recommend keeping functions short, but that rule isn't universally adhered to and naming should reflect that.

Enforcement Check length of local and non-local names. Also take function length into account.

ES.8: Avoid similar-looking names

Reason Code clarity and readability. Too-similar names slow down comprehension and increase the likelihood of error.

Example; bad

```
if (readable(i1 + l1 + ol + o1 + o0 + ol + o1 + I0 + l0)) surprise();
```

Example; bad Do not declare a non-type with the same name as a type in the same scope. This removes the need to disambiguate with a keyword such as `struct` or `enum`. It also removes a source of errors, as `struct X` can implicitly declare `X` if lookup fails.

```
struct foo { int n; };
struct foo foo();       // BAD, foo is a type already in scope
struct foo x = foo();   // requires disambiguation
```

Exception Antique header files might declare non-types and types with the same name in the same scope.

Enforcement

- Check names against a list of known confusing letter and digit combinations.
- Flag a declaration of a variable, function, or enumerator that hides a class or enumeration declared in the same scope.

ES.9: Avoid `ALL_CAPS` names

Reason Such names are commonly used for macros. Thus, `ALL_CAPS` name are vulnerable to unintended macro substitution.

Example

```
// somewhere in some header:
#define NE !=

// somewhere else in some other header:
enum Coord { N, NE, NW, S, SE, SW, E, W };

// somewhere third in some poor programmer's .cpp:
switch (direction) {
case N:
    // ...
case NE:
    // ...
// ...
}
```

Note Do not use `ALL_CAPS` for constants just because constants used to be macros.

Enforcement Flag all uses of ALL CAPS. For older code, accept ALL CAPS for macro names and flag all non-ALL-CAPS macro names.

ES.10: Declare one name (only) per declaration

Reason One-declaration-per line increases readability and avoids mistakes related to the C/C++ grammar. It also leaves room for a more descriptive end-of-line comment.

Example, bad

```
char *p, c, a[7], *pp[7], **aa[10];   // yuck!
```

Exception: a function declaration can contain several function argument declarations.

Example

```
template <class InputIterator, class Predicate>
bool any_of(InputIterator first, InputIterator last, Predicate pred);
```

or better using concepts:

```
bool any_of(InputIterator first, InputIterator last, Predicate pred);
```

Example

```
double scalbn(double x, int n);   // OK: x*pow(FLT_RADIX, n); FLT_RADIX is usually 2
```

or:

```
double scalbn(    // better: x*pow(FLT_RADIX, n); FLT_RADIX is usually 2
    double x,     // base value
    int n         // exponent
);
```

or:

```
double scalbn(double base, int exponent);   // better: base*pow(FLT_RADIX, exponent); F
```

Enforcement Flag non-function arguments with multiple declarators involving declarator operators (e.g., `int* p, q;`)

ES.11: Use `auto` to avoid redundant repetition of type names

Reason

- Simple repetition is tedious and error prone.
- When you use `auto`, the name of the declared entity is in a fixed position in the declaration, increasing readability.
- In a template function declaration the return type can be a member type.

Example Consider:

```
auto p = v.begin();   // vector<int>::iterator
auto s = v.size();
auto h = t.future();
auto q = make_unique<int[]>(s);
auto f = [](int x){ return x + 10; };
```

In each case, we save writing a longish, hard-to-remember type that the compiler already knows but a programmer could get wrong.

Example

```
template<class T>
auto Container<T>::first() -> Iterator;   // Container<T>::Iterator
```

Exception: Avoid `auto` for initializer lists and in cases where you know exactly which type you want and where an initializer might require conversion.

Example

```
auto lst = { 1, 2, 3 };   // lst is an initializer list
auto x{1};   // x is an int (after correction of the C++14 standard; initializer_list
```

Note When concepts become available, we can (and should) be more specific about the type we are deducing:

```
// ...
ForwardIterator p = algo(x, y, z);
```

Enforcement Flag redundant repetition of type names in a declaration.

ES.20: Always initialize an object

Reason Avoid used-before-set errors and their associated undefined behavior. Avoid problems with comprehension of complex initialization. Simplify refactoring.

Example

```
void use(int arg)   // bad: uninitialized variable
{
    int i;
    // ...
    i = 7;   // initialize i
}
```

No, `i = 7` does not initialize `i`; it assigns to it. Also, `i` can be read in the `...` part. Better:

```
void use(int arg)   // OK
{
    int i = 7;   // OK: initialized
    string s;    // OK: default initialized
    // ...
}
```

Note The *always initialize* rule is deliberately stronger than the *an object must be set before used* language rule. The latter, more relaxed rule, catches the technical bugs, but:

- It leads to less readable code
- It encourages people to declare names in greater than necessary scopes

- It leads to harder to read code
- It leads to logic bugs by encouraging complex code
- It hampers refactoring

The *always initialize* rule is a style rule aimed to improve maintainability as well as a rule protecting against used-before-set errors.

Example Here is an example that is often considered to demonstrate the need for a more relaxed rule for initialization

```
widget i;    // "widget" a type that's expensive to initialize, possibly a large POD
widget j;

if (cond) {  // bad: i and j are initialized "late"
    i = f1();
    j = f2();
}
else {
    i = f3();
    j = f4();
}
```

This cannot trivially be rewritten to initialize `i` and `j` with initializers. Note that for types with a default constructor, attempting to postpone initialization simply leads to a default initialization followed by an assignment. A popular reason for such examples is "efficiency", but a compiler that can detect whether we made a used-before-set error can also eliminate any redundant double initialization.

At the cost of repeating `cond` we could write:

```
widget i = (cond) ? f1() : f3();
widget j = (cond) ? f2() : f4();
```

Assuming that there is a logical connection between `i` and `j`, that connection should probably be expressed in code:

```
pair<widget,widget> make_related_widgets(bool x)
{
    return (x) ? {f1(),f2()} : {f3(),f4() };
}

auto init = make_related_widgets(cond);
widget i = init.first;
widget j = init.second;
```

Obviously, what we really would like is a construct that initialized n variables from a `tuple`. For example:

```
auto {i,j} = make_related_widgets(cond);    // Not C++14
```

Today, we might approximate that using `tie()`:

```
widget i;       // bad: uninitialized variable
widget j;
tie(i,j) = make_related_widgets(cond);
```

This may be seen as an example of the *immediately initialize from input* exception below.

Creating optimal and equivalent code from all of these examples should be well within the capabilities of modern C++ compilers (but don't make performance claims without measuring; a compiler may very well not generate optimal code for every example and there may be language rules preventing some optimization that you would have liked in a particular case).

Note Complex initialization has been popular with clever programmers for decades. It has also been a major source of errors and complexity. Many such errors are introduced during maintenance years after the initial implementation.

Exception It you are declaring an object that is just about to be initialized from input, initializing it would cause a double initialization. However, beware that this may leave uninitialized data beyond the input - and that has been a fertile source of errors and security breaches:

```
constexpr int max = 8*1024;
int buf[max];           // OK, but suspicious: uninitialized
f.read(buf, max);
```

The cost of initializing that array could be significant in some situations. However, such examples do tend to leave uninitialized variables accessible, so they should be treated with suspicion.

```
constexpr int max = 8*1024;
int buf[max] = {0};   // better in some situations
f.read(buf, max);
```

When feasible use a library function that is known not to overflow. For example:

```
string s;   // s is default initialized to ""
cin >> s;   // s expands to hold the string
```

Don't consider simple variables that are targets for input operations exceptions to this rule:

```
int i;   // bad
// ...
cin >> i;
```

In the not uncommon case where the input target and the input operation get separated (as they should not) the possibility of used-before-set opens up.

```
int i2 = 0;   // better
// ...
cin >> i;
```

A good optimizer should know about input operations and eliminate the redundant operation.

Example Using an `uninitialized` or sentinel value is a symptom of a problem and not a solution:

```
widget i = uninit;  // bad
widget j = uninit;

// ...
use(i);         // possibly used before set
// ...

if (cond) {     // bad: i and j are initialized "late"
    i = f1();
    j = f2();
}
else {
    i = f3();
    j = f4();
}
```

Now the compiler cannot even simply detect a used-before-set. Further, we've introduced complexity in the state space for widget: which operations are valid on an `unint` widget and which are not?

Note Sometimes, a lambda can be used as an initializer to avoid an uninitialized variable:

```
error_code ec;
Value v = [&] {
    auto p = get_value();   // get_value() returns a pair<error_code, Value>
    ec = p.first;
    return p.second;
}();
```

or maybe:

```
Value v = [] {
    auto p = get_value();   // get_value() returns a pair<error_code, Value>
    if (p.first) throw Bad_value{p.first};
    return p.second;
}();
```

See also: ES.28

Enforcement

- Flag every uninitialized variable. Don't flag variables of user-defined types with default constructors.
- Check that an uninitialized buffer is written into *immediately* after declaration. Passing an uninitialized variable as a reference to non-`const` argument can be assumed to be a write into the variable.

ES.21: Don't introduce a variable (or constant) before you need to use it

Reason Readability. To limit the scope in which the variable can be used.

Example

```
int x = 7;
// ... no use of x here ...
++x;
```

Enforcement Flag declarations that are distant from their first use.

ES.22: Don't declare a variable until you have a value to initialize it with

Reason Readability. Limit the scope in which a variable can be used. Don't risk used-before-set. Initialization is often more efficient than assignment.

Example, bad

```
string s;
// ... no use of s here ...
s = "what a waste";
```

Example, bad

```
SomeLargeType var;   // ugly CaMeLcAsEvArIaBlE

if (cond)   // some non-trivial condition
    Set(&var);
else if (cond2 || !cond3) {
    var = Set2(3.14);
}
else {
    var = 0;
    for (auto& e : something)
        var += e;
}

// use var; that this isn't done too early can be enforced statically with only contro
```

This would be fine if there was a default initialization for `SomeLargeType` that wasn't too expensive. Otherwise, a programmer might very well wonder if every possible path through the maze of conditions has been covered. If not, we have a "use before set" bug. This is a maintenance trap.

For initializers of moderate complexity, including for `const` variables, consider using a lambda to express the initializer; see ES.28.

Enforcement

- Flag declarations with default initialization that are assigned to before they are first read.
- Flag any complicated computation after an uninitialized variable and before its use.

ES.23: Prefer the {} initializer syntax

Reason The rules for `{}` initialization are simpler, more general, less ambiguous, and safer than for other forms of initialization.

Example

```
int x {f(99)};
vector<int> v = {1, 2, 3, 4, 5, 6};
```

Exception For containers, there is a tradition for using `{...}` for a list of elements and `(...)` for sizes:

```
vector<int> v1(10);    // vector of 10 elements with the default value 0
vector<int> v2 {10};   // vector of 1 element with the value 10
```

Note `{}`-initializers do not allow narrowing conversions.

Example

```
int x {7.9};   // error: narrowing
int y = 7.9;   // OK: y becomes 7. Hope for a compiler warning
```

Note `{}` initialization can be used for all initialization; other forms of initialization can't:

```
auto p = new vector<int> {1, 2, 3, 4, 5};   // initialized vector
D::D(int a, int b) :m{a, b} {   // member initializer (e.g., m might be a pair)
    // ...
};
X var {};   // initialize var to be empty
struct S {
    int m {7};   // default initializer for a member
    // ...
};
```

Note Initialization of a variable declared using `auto` with a single value, e.g., `{v}`, had surprising results until recently:

```
auto x1 {7};        // x1 is an int with the value 7
auto x2 = {7};      // x2 is an initializer_list<int> with an element 7

auto x11 {7, 8};    // error: two initializers
auto x22 = {7, 8};  // x2 is an initializer_list<int> with elements 7 and 8
```

Exception Use `={...}` if you really want an `initializer_list<T>`

```
auto fib10 = {0, 1, 2, 3, 5, 8, 13, 25, 38, 63};   // fib10 is a list
```

Example

```
template<typename T>
void f()
{
    T x1(1);     // T initialized with 1
    T x0();      // bad: function declaration (often a mistake)

    T y1 {1};    // T initialized with 1
    T y0 {};     // default initialized T
    // ...
}
```

See also: Discussion

Enforcement Tricky.

- Don't flag uses of `=` for simple initializers.
- Look for `=` after `auto` has been seen.

ES.24: Use a `unique_ptr<T>` to hold pointers

Reason Using `std::unique_ptr` is the simplest way to avoid leaks. It is reliable, it makes the type system do much of the work to validate ownership safety, it increases readability, and it has zero or near zero runtime cost.

Example

```
void use(bool leak)
{
    auto p1 = make_unique<int>(7);   // OK
    int* p2 = new int{7};            // bad: might leak
    // ...
    if (leak) return;
    // ...
}
```

If `leak == true` the object pointed to by `p2` is leaked and the object pointed to by `p1` is not.

Enforcement Look for raw pointers that are targets of `new`, `malloc()`, or functions that may return such pointers.

ES.25: Declare objects `const` or `constexpr` unless you want to modify its value later on

Reason That way you can't change the value by mistake. That way may offer the compiler optimization opportunities.

Example

```
void f(int n)
{
    const int bufmax = 2 * n + 2;  // good: we can't change bufmax by accident
    int xmax = n;                  // suspicious: is xmax intended to change?
    // ...
}
```

Enforcement Look to see if a variable is actually mutated, and flag it if not. Unfortunately, it may be impossible to detect when a non-`const` was not *intended* to vary (vs when it merely did not vary).

ES.26: Don't use a variable for two unrelated purposes

Reason Readability.

Example, bad

```
void use()
{
    int i;
    for (i = 0; i < 20; ++i) { /* ... */ }
    for (i = 0; i < 200; ++i) { /* ... */ } // bad: i recycled
}
```

Enforcement Flag recycled variables.

ES.27: Use `std::array` or `stack_array` for arrays on the stack

Reason They are readable and don't implicitly convert to pointers. They are not confused with non-standard extensions of built-in arrays.

Example, bad

```
const int n = 7;
int m = 9;

void f()
{
    int a1[n];
    int a2[m];   // error: not ISO C++
    // ...
}
```

Note The definition of `a1` is legal C++ and has always been. There is a lot of such code. It is error-prone, though, especially when the bound is non-local. Also, it is a "popular" source of errors (buffer overflow, pointers from array decay, etc.). The definition of `a2` is C but not C++ and is considered a security risk

Example

```
const int n = 7;
int m = 9;

void f()
{
    array<int, n> a1;
    stack_array<int> a2(m);
    // ...
}
```

Enforcement

- Flag arrays with non-constant bounds (C-style VLAs)
- Flag arrays with non-local constant bounds

ES.28: Use lambdas for complex initialization, especially of `const` variables

Reason It nicely encapsulates local initialization, including cleaning up scratch variables needed only for the initialization, without needing to create a needless nonlocal yet non-reusable function. It also works for variables that should be `const` but only after some initialization work.

Example, bad

```
widget x;   // should be const, but:
for (auto i = 2; i <= N; ++i) {             // this could be some
    x += some_obj.do_something_with(i);  // arbitrarily long code
}                                        // needed to initialize x
// from here, x should be const, but we can't say so in code in this style
```

Example, good

```
const widget x = [&]{
    widget val;                                // assume that widget has a default cons
    for (auto i = 2; i <= N; ++i) {            // this could be some
        val += some_obj.do_something_with(i);  // arbitrarily long code
    }                                          // needed to initialize x
    return val;
}();
```

Example

```
string var = [&]{
    if (!in) return "";   // default
    string s;
    for (char c : in >> c)
        s += toupper(c);
    return s;
}(); // note ()
```

If at all possible, reduce the conditions to a simple set of alternatives (e.g., an `enum`) and don't mix up selection and initialization.

Example

```
owner<istream&> in = [&]{
    switch (source) {
    case default:       owned=false; return cin;
    case command_line:  owned=true;  return *new istringstream{argv[2]};
    case file:          owned=true;  return *new ifstream{argv[2]};
}();
```

Enforcement Hard. At best a heuristic. Look for an uninitialized variable followed by a loop assigning to it.

ES.30: Don't use macros for program text manipulation

Reason Macros are a major source of bugs. Macros don't obey the usual scope and type rules. Macros ensure that the human reader see something different from what the compiler sees. Macros complicates tool building.

Example, bad

```
#define Case break; case   /* BAD */
```

This innocuous-looking macro makes a single lower case `c` instead of a `C` into a bad flow-control bug.

Note This rule does not ban the use of macros for "configuration control" use in `#ifdef`s, etc.

Enforcement Scream when you see a macro that isn't just use for source control (e.g., `#ifdef`)

ES.31: Don't use macros for constants or "functions"

Reason Macros are a major source of bugs. Macros don't obey the usual scope and type rules. Macros don't obey the usual rules for argument passing. Macros ensure that the human reader sees something different from what the compiler sees. Macros complicate tool building.

Example, bad

```
#define PI 3.14
#define SQUARE(a, b) (a*b)
```

Even if we hadn't left a well-known bug in `SQUARE` there are much better behaved alternatives; for example:

```
constexpr double pi = 3.14;
template<typename T> T square(T a, T b) { return a*b; }
```

Enforcement Scream when you see a macro that isn't just used for source control (e.g., `#ifdef`)

ES.32: Use `ALL_CAPS` for all macro names

Reason Convention. Readability. Distinguishing macros.

Example

```
#define forever for(;;)   /* very BAD */

#define FOREVER for(;;)   /* Still evil, but at least visible to humans */
```

Enforcement Scream when you see a lower case macro.

ES.33: If you must use macros, give them unique names

Reason Macros do not obey scope rules.

Example

```
#define MYCHAR        /* BAD, will eventually clash with someone else's MYCHAR*/

#define ZCORP_CHAR    /* Still evil, but less likely to clash */
```

Note Avoid macros if you can: ES.30, ES.31, and ES.32. However, there are billions of lines of code littered with macros and a long tradition for using and overusing macros. If you are forced to use macros, use long names and supposedly unique prefixes (e.g., your organization's name) to lower the likelihood of a clash.

Enforcement Warn against short macro names.

ES.40: Don't define a (C-style) variadic function

Reason Not type safe. Requires messy cast-and-macro-laden code to get working right.

Example

```
??? <vararg>
```

Alternative: Overloading. Templates. Variadic templates.

Note There are rare used of variadic functions in SFINAE code, but those don't actually run and don't need the `<vararg>` implementation mess.

Enforcement Flag definitions of C-style variadic functions.

ES.stmt: Statements

Statements control the flow of control (except for function calls and exception throws, which are expressions).

ES.70: Prefer a `switch`-statement to an `if`-statement when there is a choice

Reason

- Readability.
- Efficiency: A `switch` compares against constants and is usually better optimized than a series of tests in an `if-then-else` chain.
- a `switch` is enables some heuristic consistency checking. For example, have all values of an `enum` been covered? If not, is there a `default`?

Example

```
void use(int n)
{
    switch (n) {   // good
    case 0:   // ...
    case 7:   // ...
    }
}
```

rather than:

```
void use2(int n)
{
    if (n == 0)   // bad: if-then-else chain comparing against a set of constants
        // ...
    else if (n == 7)
        // ...
}
```

Enforcement Flag if-then-else chains that check against constants (only).

ES.71: Prefer a range-`for`-statement to a `for`-statement when there is a choice

Reason Readability. Error prevention. Efficiency.

Example

```
for (int i = 0; i < v.size(); ++i)   // bad
        cout << v[i] << '\n';

for (auto p = v.begin(); p != v.end(); ++p)   // bad
    cout << *p << '\n';

for (auto& x : v)    // OK
    cout << x << '\n';

for (int i = 1; i < v.size(); ++i) // touches two elements: can't be a range-for
    cout << v[i] + v[i-1] << '\n';

for (int i = 0; i < v.size(); ++i) // possible side-effect: can't be a range-for
    cout << f(v, &v[i]) << '\n';

for (int i = 0; i < v.size(); ++i) { // body messes with loop variable: can't be a rang
    if (i % 2)
        ++i;   // skip even elements
    else
        cout << v[i] << '\n';
}
```

A human or a good static analyzer may determine that there really isn't a side effect on `v` in `f(v, &v[i])` so that the loop can be rewritten.

"Messing with the loop variable" in the body of a loop is typically best avoided.

Note Don't use expensive copies of the loop variable of a range-`for` loop:

```
for (string s : vs) // ...
```

This will copy each elements of `vs` into `s`. Better:

```
for (string& s : vs) // ...
```

Better still, if the loop variable isn't modified or copied:

```
for (const string& s : vs) // ...
```

Enforcement Look at loops, if a traditional loop just looks at each element of a sequence, and there are no side-effects on what it does with the elements, rewrite the loop to a ranged-for loop.

ES.72: Prefer a `for`-statement to a `while`-statement when there is an obvious loop variable

Reason Readability: the complete logic of the loop is visible "up front". The scope of the loop variable can be limited.

Example

```
for (int i = 0; i < vec.size(); i++) {
 // do work
}
```

Example, bad

```
int i = 0;
while (i < vec.size()) {
 // do work
 i++;
}
```

Enforcement ???

ES.73: Prefer a `while`-statement to a `for`-statement when there is no obvious loop variable

Reason ???

Example

```
???
```

Enforcement ???

ES.74: Prefer to declare a loop variable in the initializer part of as `for`-statement

Reason Limit the loop variable visibility to the scope of the loop. Avoid using the loop variable for other purposes after the loop.

Example

```
for (int i = 0; i < 100; ++i) {   // GOOD: i var is visible only inside the loop
    // ...
}
```

Example, don't

```
int j;                            // BAD: j is visible outside the loop
for (j = 0; j < 100; ++j) {
    // ...
}
// j is still visible here and isn't needed
```

See also: Don't use a variable for two unrelated purposes

Enforcement Warn when a variable modified inside the `for`-statement is declared outside the loop and not being used outside the loop.

Discussion: Scoping the loop variable to the loop body also helps code optimizers greatly. Recognizing that the induction variable is only accessible in the loop body unblocks optimizations such as hoisting, strength reduction, loop-invariant code motion, etc.

ES.75: Avoid `do`-statements

Reason Readability, avoidance of errors. The termination condition is at the end (where it can be overlooked) and the condition is not checked the first time through. ???

Example

```
int x;
do {
    cin >> x;
    x
} while (x < 0);
```

Enforcement ???

ES.76: Avoid `goto`

Reason Readability, avoidance of errors. There are better control structures for humans; `goto` is for machine generated code.

Exception Breaking out of a nested loop. In that case, always jump forwards.

Example

```
???
```

Example There is a fair amount of use of the C goto-exit idiom:

```
void f()
{
    // ...
        goto exit;
    // ...
        goto exit;
    // ...
exit:
    ... common cleanup code ...
}
```

This is an ad-hoc simulation of destructors. Declare your resources with handles with destructors that clean up.

Enforcement

- Flag `goto`. Better still flag all `goto`s that do not jump from a nested loop to the statement immediately after a nest of loops.

ES.77: ??? `continue`

Reason ???

Example

```
???
```

Enforcement ???

ES.78: Always end a non-empty `case` with a `break`

Reason Accidentally leaving out a `break` is a fairly common bug. A deliberate fallthrough is a maintenance hazard.

Example

```
switch(eventType)
{
case Information:
    update_status_bar();
    break;
case Warning:
    write_event_log();
case Error:
    display_error_window(); // Bad
    break;
}
```

It is easy to overlook the fallthrough. Be explicit:

```
switch(eventType)
{
case Information:
    update_status_bar();
    break;
case Warning:
    write_event_log();
    // fall through
case Error:
    display_error_window(); // Bad
    break;
}
```

There is a proposal for a `[[fallthrough]]` annotation.

Note Multiple case labels of a single statement is OK:

```
switch (x) {
case 'a':
case 'b':
case 'f':
    do_something(x);
    break;
}
```

Enforcement Flag all fall throughs from non-empty `case`s.

ES.79: ??? `default`

Reason ???

Example

```
???
```

Enforcement ???

ES.85: Make empty statements visible

Reason Readability.

Example

```
for (i = 0; i < max; ++i);   // BAD: the empty statement is easily overlooked
v[i] = f(v[i]);

for (auto x : v) {           // better
    // nothing
}
v[i] = f(v[i]);
```

Enforcement Flag empty statements that are not blocks and don't contain comments.

ES.86: Avoid modifying loop control variables inside the body of raw for-loops

Reason The loop control up front should enable correct reasoning about what is happening inside the loop. Modifying loop counters in both the iteration-expression and inside the body of the loop is a perennial source of surprises and bugs.

Example

```
for (int i=0; i<10; ++i) {
    // no updates to i -- ok
}

for (int i=0; i<10; ++i) {
    //
    if (/* something */) ++i; // BAD
```

```
    //
}

bool skip=false;
for (int i=0; i<10; ++i) {
    if (skip) { skip = false; continue; }
    //
    if (/* something */) skip = true;  // Better: using two variable for two concepts.
    //
}
```

Enforcement Flag variables that are potentially updated (have a non-const use) in both the loop control iteration-expression and the loop body.

ES.expr: Expressions

Expressions manipulate values.

ES.40: Avoid complicated expressions

Reason Complicated expressions are error-prone.

Example

```
while ((c = getc()) != -1)   // bad: assignment hidden in subexpression

while ((cin >> c1, cin >> c2), c1 == c2) // bad: two non-local variables assigned in a

for (char c1, c2; cin >> c1 >> c2 && c1 == c2;)   // better, but possibly still too com

int x = ++i + ++j;    // OK: iff i and j are not aliased

v[i] = v[j] + v[k];   // OK: iff i != j and i != k

x = a + (b = f()) + (c = g()) * 7;   // bad: multiple assignments "hidden" in subexpres

x = a & b + c * d && e ^ f == 7;     // bad: relies on commonly misunderstood precedenc

x = x++ + x++ + ++x;   // bad: undefined behavior
```

Some of these expressions are unconditionally bad (e.g., they rely on undefined behavior). Others are simply so complicated and/or unusual that even good programmers could misunderstand them or overlook a problem when in a hurry.

Note A programmer should know and use the basic rules for expressions.

Example

```
x=k * y + z;             // OK

auto t1 = k*y;           // bad: unnecessarily verbose
x = t1 + z;

if (0 <= x && x < max)   // OK

auto t1 = 0 <= x;        // bad: unnecessarily verbose
auto t2 = x < max;
if (t1 && t2)            // ...
```

Enforcement Tricky. How complicated must an expression be to be considered complicated? Writing computations as statements with one operation each is also confusing. Things to consider:

- side effects: side effects on multiple non-local variables (for some definition of non-local) can be suspect, especially if the side effects are in separate subexpressions
- writes to aliased variables
- more than N operators (and what should N be?)
- reliance of subtle precedence rules
- uses undefined behavior (can we catch all undefined behavior?)
- implementation defined behavior?
- ???

ES.41: If in doubt about operator precedence, parenthesize

Reason Avoid errors. Readability. Not everyone has the operator table memorized.

Example

```
const unsigned int flag = 2;
unsigned int a = flag;

if (a & flag != 0)  // bad: means a&(flag != 0)
```

Note: We recommend that programmers know their precedence table for the arithmetic operations, the logical operations, but consider mixing bitwise logical operations with other operators in need of parentheses.

```
if ((a & flag) != 0)  // OK: works as intended
```

Note You should know enough not to need parentheses for:

```
if (a < 0 || a <= max) {
    // ...
}
```

Enforcement

- Flag combinations of bitwise-logical operators and other operators.
- Flag assignment operators not as the leftmost operator.
- ???

ES.42: Keep use of pointers simple and straightforward

Reason Complicated pointer manipulation is a major source of errors.

- Do all pointer arithmetic on a `span` (exception ++p in simple loop???)
- Avoid pointers to pointers
- ???

Example

```
???
```

Enforcement We need a heuristic limiting the complexity of pointer arithmetic statement.

ES.43: Avoid expressions with undefined order of evaluation

Reason You have no idea what such code does. Portability. Even if it does something sensible for you, it may do something different on another compiler (e.g., the next release of your compiler) or with a different optimizer setting.

Example

```
v[i] = ++i;    //  the result is undefined
```

A good rule of thumb is that you should not read a value twice in an expression where you write to it.

Example

```
???
```

Note What is safe?

Enforcement Can be detected by a good analyzer.

ES.44: Don't depend on order of evaluation of function arguments

Reason Because that order is unspecified.

Example

```
int i = 0;
f(++i, ++i);
```

The call will most likely be `f(0, 1)` or `f(1, 0)`, but you don't know which. Technically, the behavior is undefined.

Example ??? overloaded operators can lead to order of evaluation problems (shouldn't :-()

```
f1()->m(f2());    // m(f1(), f2())
cout << f1() << f2();    // operator<<(operator<<(cout, f1()), f2())
```

Enforcement Can be detected by a good analyzer.

ES.45: Avoid "magic constants"; use symbolic constants

Reason Unnamed constants embedded in expressions are easily overlooked and often hard to understand:

Example

```
for (int m = 1; m <= 12; ++m)    // don't: magic constant 12
    cout << month[m] << '\n';
```

No, we don't all know that there are 12 months, numbered 1..12, in a year. Better:

```
constexpr int month_count = 12;   // months are numbered 1..12

for (int m = first_month; m <= month_count; ++m)   // better
    cout << month[m] << '\n';
```

Better still, don't expose constants:

```
for (auto m : month)
    cout << m << '\n';
```

Enforcement Flag literals in code. Give a pass to `0`, `1`, `nullptr`, `\n`, `""`, and others on a positive list.

ES.46: Avoid lossy (narrowing, truncating) arithmetic conversions

Reason A narrowing conversion destroys information, often unexpectedly so.

Example, bad A key example is basic narrowing:

```
double d = 7.9;
int i = d;    // bad: narrowing: i becomes 7
i = (int)d;   // bad: we're going to claim this is still not explicit enough

void f(int x, long y, double d)
{
    char c1 = x;   // bad: narrowing
    char c2 = y;   // bad: narrowing
    char c3 = d;   // bad: narrowing
}
```

Note The guideline support library offers a `narrow` operation for specifying that narrowing is acceptable and a `narrow` ("narrow if") that throws an exception if a narrowing would throw away information:

```
i = narrow_cast<int>(d);   // OK (you asked for it): narrowing: i becomes 7
i = narrow<int>(d);        // OK: throws narrowing_error
```

We also include lossy arithmetic casts, such as from a negative floating point type to an unsigned integral type:

```
double d = -7.9;
unsigned u = 0;

u = d;                              // BAD
u = narrow_cast<unsigned>(d);       // OK (you asked for it): u becomes 0
u = narrow<unsigned>(d);            // OK: throws narrowing_error
```

Enforcement A good analyzer can detect all narrowing conversions. However, flagging all narrowing conversions will lead to a lot of false positives. Suggestions:

- flag all floating-point to integer conversions (maybe only float->char and double->int. Here be dragons! we need data)
- flag all long->char (I suspect int->char is very common. Here be dragons! we need data)
- consider narrowing conversions for function arguments especially suspect

ES.47: Use `nullptr` rather than `0` or `NULL`

Reason Readability. Minimize surprises: `nullptr` cannot be confused with an `int`. `nullptr` also has a well-specified (very restrictive) type, and thus works in more scenarios where type deduction might do the wrong thing on `NULL` or `0`.

Example Consider:

```
void f(int);
void f(char*);
f(0);         // call f(int)
f(nullptr);   // call f(char*)
```

Enforcement Flag uses of `0` and `NULL` for pointers. The transformation may be helped by simple program transformation.

ES.48: Avoid casts

Reason Casts are a well-known source of errors. Makes some optimizations unreliable.

Example

```
???
```

Note Programmer who write casts typically assumes that they know what they are doing. In fact, they often disable the general rules for using values. Overload resolution and template instantiation usually pick the right function if there is a right function to pick. If there is not, maybe there ought to be, rather than applying a local fix (cast).

Note Casts are necessary in a systems programming language. For example, how else would we get the address of a device register into a pointer? However, casts are seriously overused as well as a major source of errors.

Note If you feel the need for a lot of casts, there may be a fundamental design problem.

Enforcement

- Force the elimination of C-style casts
- Warn against named casts
- Warn if there are many functional style casts (there is an obvious problem in quantifying 'many').

ES.49: If you must use a cast, use a named cast

Reason Readability. Error avoidance. Named casts are more specific than a C-style or functional cast, allowing the compiler to catch some errors.

The named casts are:

- `static_cast`
- `const_cast`
- `reinterpret_cast`
- `dynamic_cast`
- `std::move` // `move(x)` is an rvalue reference to `x`
- `std::forward` // `forward(x)` is an rvalue reference to `x`
- `gsl::narrow_cast` // `narrow_cast<T>(x)` is `static_cast<T>(x)`
- `gsl::narrow` // `narrow<T>(x)` is `static_cast<T>(x)` if `static_cast<T>(x) == x` or it throws `narrowing_error`

Example

```
???
```

Note When converting between types with no information loss (e.g. from float to double or int64 from int32), brace initialization may be used instead.

```
double d{some_float};
int64_t i{some_int32};
```

This makes it clear that the type conversion was intended and also prevents conversions between types that might result in loss of precision. (It is a compilation error to try to initialize a float from a double in this fashion, for example.)

Enforcement Flag C-style and functional casts.

ES.50: Don't cast away `const`

Reason It makes a lie out of `const`.

Note Usually the reason to "cast away `const`" is to allow the updating of some transient information of an otherwise immutable object. Examples are caching, memoization, and precomputation. Such examples are often handled as well or better using `mutable` or an indirection than with a `const_cast`.

Example

```
???
```

Enforcement Flag `const_casts`.

ES.55: Avoid the need for range checking

Reason Constructs that cannot overflow do not overflow (and usually run faster):

Example

```
for (auto& x : v)      // print all elements of v
    cout << x << '\n';

auto p = find(v, x);   // find x in v
```

Enforcement Look for explicit range checks and heuristically suggest alternatives.

ES.56: Write `std::move()` only when you need to explicitly move an object to another scope

Reason We move, rather than copy, to avoid duplication and for improved performance.

A move typically leaves behind an empty object (C.64), which can be surprising or even dangerous, so we try to avoid moving from lvalues (they might be accessed later).

Notes Moving is done implicitly when the source is an rvalue (e.g., value in a `return` treatment or a function result), so don't pointlessly complicate code in those cases by writing `move` explicitly. Instead, write short functions that return values, and both the function's return and the caller's accepting of the return will be optimized naturally.

In general, following the guidelines in this document (including not making variables' scopes needlessly large, writing short functions that return values, returning local variables) help eliminate most need for explicit `std::move`.

Explicit `move` is needed to explicitly move an object to another scope, notably to pass it to a "sink" function and in the implementations of the move operations themselves (move constructor, move assignment operator) and swap operations.

Example, bad

```
void sink(X&& r);   // sink takes ownership of r

void user()
{
    X x;
    sink(r);              // error: cannot bind an lvalue to a rvalue reference
    sink(std::move(r));   // OK: sink takes the contents of r, r must now assumed to be
    // ...
    use(r);               // probably a mistake
}
```

Usually, a `std::move()` is used as an argument to a `&&` parameter. And after you do that, assume the object has been moved from (see C.64) and don't read its state again until you first set it to a new value.

```
void f() {
    string s1 = "supercalifragilisticexpialidocious";

    string s2 = s1;             // ok, takes a copy
    assert(s1=="supercalifragilisticexpialidocious");  // ok

    string s3 = move(s1);       // bad, if you want to keep using s1's value
    assert(s1=="supercalifragilisticexpialidocious");  // bad, assert will likely fail,
}
```

Example

```
void sink( unique_ptr<widget> p );  // pass ownership of p to sink()

void f() {
    auto w = make_unique<widget>();
    // ...
    sink( std::move(w) );               // ok, give to sink()
    // ...
    sink(w);    // Error: unique_ptr is carefully designed so that you cannot copy it
}
```

Notes `std::move()` is a cast to `&&` in disguise; it doesn't itself move anything, but marks a named object as a candidate that can be moved from. The language already knows the common cases where objects can be moved from, especially when returning values from functions, so don't complicate code with redundant 'std::move()'s.

Never write `std::move()` just because you've heard "it's more efficient." In general, don't believe claims of "efficiency" without data (???). In general, don't complicate your code without reason (??)

Example, bad

```
vector<int> make_vector() {
    vector<int> result;
    // ... load result with data
    return std::move(result);       // bad; just write "return result;"
}
```

Never write `return move(local_variable);`, because the language already knows the variable is a move candidate. Writing `move` in this code won't help, and can actually be detrimental because on some compilers it interferes with RVO (the return value optimization) by creating an additional reference alias to the local variable.

Example, bad

```
vector<int> v = std::move(make_vector());   // bad; the std::move is entirely redundant
```

Never write `move` on a returned value such as `x = move(f());` where `f` returns by value. The language already knows that a returned value is a temporary object that can be moved from.

Example

```
void mover(X&& x) {
    call_something( std::move(x) );         // ok
    call_something( std::forward<X>(x) );   // bad, don't std::forward an rvalue refere
    call_something( x );                    // suspicious, why not std::move?
}

template<class T>
void forwarder(T&& t) {
    call_something( std::move(t) );         // bad, don't std::move a forwarding refere
    call_something( std::forward<T>(t) );   // ok
    call_something( t );                    // suspicious, why not std::forward?
}
```

Enforcement

- Flag use of `std::move(x)` where `x` is an rvalue or the language will already treat it as an rvalue, including `return std::move(local_variable);` and `std::move(f())` on a function that returns by value.
- Flag functions taking an `S&&` parameter if there is no `const S&` overload to take care of lvalues.
- Flag a `std::move`s argument passed to a parameter, except when the parameter type is one of the following: an `X&&` rvalue reference; a `T&&` forwarding reference where `T` is a template parameter type; or by value and the type is move-only.
- Flag when `std::move` is applied to a forwarding reference (`T&&` where `T` is a template parameter type). Use `std::forward` instead.
- Flag when `std::move` is applied to other than an rvalue reference. (More general case of the previous rule to cover the non-forwarding cases.)
- Flag when `std::forward` is applied to an rvalue reference (`X&&` where `X` is a concrete type). Use `std::move` instead.
- Flag when `std::forward` is applied to other than a forwarding reference. (More general case of the previous rule to cover the non-moving cases.)
- Flag when an object is potentially moved from and the next operation is a `const` operation; there should first be an intervening non-`const` operation, ideally assignment, to first reset the object's value.

ES.60: Avoid `new` and `delete` outside resource management functions

Reason Direct resource management in application code is error-prone and tedious.

Note also known as "No naked `new`!"

Example, bad

```
void f(int n)
{
    auto p = new X[n];   // n default constructed Xs
    // ...
    delete[] p;
}
```

There can be code in the `...` part that causes the `delete` never to happen.

See also: R: Resource management.

Enforcement Flag naked `new`s and naked `delete`s.

ES.61: delete arrays using `delete[]` and non-arrays using `delete`

Reason That's what the language requires and mistakes can lead to resource release errors and/or memory corruption.

Example, bad

```
void f(int n)
{
    auto p = new X[n];   // n default constructed Xs
    // ...
    delete p;   // error: just delete the object p, rather than delete the array p[]
}
```

Note This example not only violates the no naked `new` rule as in the previous example, it has many more problems.

Enforcement

- if the `new` and the `delete` is in the same scope, mistakes can be flagged.
- if the `new` and the `delete` are in a constructor/destructor pair, mistakes can be flagged.

ES.62: Don't compare pointers into different arrays

Reason The result of doing so is undefined.

Example, bad

```
void f(int n)
{
    int a1[7];
    int a2[9];
    if (&a1[5] < &a2[7]) {}       // bad: undefined
    if (0 < &a1[5] - &a2[7]) {}   // bad: undefined
}
```

Note This example has many more problems.

Enforcement ???

ES.63: Don't slice

Reason Slicing - that is, copying only part of an object using assignment or initialization - most often leads to errors because the object was meant to be considered as a whole. In the rare cases where the slicing was deliberate the code can be surprising.

Example

```
class Shape { /* ... */ };
class Circle : public Shape { /* ... */ Point c; int r; };

Circle c {{0,0}, 42};
Shape s {c};    // copy Shape part of Circle
```

The result will be meaningless because the center and radius will not be copied from `c` into `s`. The first defense against this is to define the base class `Shape` not to allow this.

Alternative If you mean to slice, define an explicit operations to do so. This saves readers from confusion. For example:

```
class Smiley : public Circle {
    public:
    Circle copy_circle();
    // ...
};

Smiley sm { /* ... */ };
Circle c1 {sm};  // ideally prevented by the definition of Circle
Circle c2 {sm.copy_circle()};
```

Enforcement Warn against slicing.

Arithmetic

ES.100: Don't mix signed and unsigned arithmetic

Reason Avoid wrong results.

Example

```
unsigned x = 100;
unsigned y = 102;
cout << abs(x-y) << '\n'; // wrong result
```

Note Unfortunately, C++ uses signed integers for array subscripts and the standard library uses unsigned integers for container subscripts. This precludes consistency.

Enforcement Compilers already know and sometimes warn.

ES.101: Use unsigned types for bit manipulation

Reason Unsigned types support bit manipulation without surprises from sign bits.

Example

```
???
```

Exception: Use unsigned types if you really want modulo arithmetic - add comments as necessary noting the reliance on overflow behavior, as such code is going to be surprising for many programmers.

Enforcement ???

ES.102: Use signed types for arithmetic

Reason Signed types support modulo arithmetic without surprises from lack of sign bits.

Example

```
???
```

Exception: Use unsigned types if you really want modulo arithmetic - add comments as necessary noting the reliance on overflow behavior, as such code is going to be surprising for many programmers.

Enforcement ???

ES.103: Don't overflow

Reason Overflow usually makes your numeric algorithm meaningless. Incrementing a value beyond a maximum value can lead to memory corruption and undefined behavior.

Example, bad

```
int a[10];
a[10] = 7;   // bad

int n = 0;
while (n++ < 10)
    a[n - 1] = 9; // bad (twice)
```

Example, bad

```
int n = numeric_limits<int>::max();
int m = n + 1;   // bad
```

Example, bad

```
int area(int h, int w) { return h * w; }

auto a = area(10'000'000, 100'000'000);   // bad
```

Exception: Use unsigned types if you really want modulo arithmetic.

Alternative: For critical applications that can afford some overhead, use a range-checked integer and/or floating-point type.

Enforcement ???

ES.104: Don't underflow

Reason Decrementing a value beyond a minimum value can lead to memory corruption and undefined behavior.

Example, bad

```
int a[10];
a[-2] = 7;   // bad

int n = 101;
while (n--)
    a[n - 1] = 9;   // bad (twice)
```

Exception: Use unsigned types if you really want modulo arithmetic.

Enforcement ???

ES.105: Don't divide by zero

Reason The result is undefined and probably a crash.

Note This also applies to `%`.

Example; bad

```
double divide(int a, int b) {
    return a/b;                    // BAD, should be checked (e.g., in a precondition)
}
```

Example; good

```
double divide(int a, int b) {
    Expects(b != 0);               // good, address via precondition (and replace with cor
    return a/b;
}

double divide(int a, int b) {
    return b ? a/b : quiet_NaN<double>();       // good, address via check
}
```

Alternative: For critical applications that can afford some overhead, use a range-checked integer and/or floating-point type.

Enforcement

- Flag division by an integral value that could be zero

PER: Performance

??? should this section be in the main guide???

This section contains rules for people who needs high performance or low-latency. That is, rules that relates to how to use as little time and as few resources as possible to achieve a task in a predictably short time. The rules in this section are more restrictive and intrusive than what is needed for many (most) applications. Do not blindly try to follow them in general code because achieving the goals of low latency requires extra work.

Performance rule summary:

- PER.1: Don't optimize without reason
- PER.2: Don't optimize prematurely
- PER.3: Don't optimize something that's not performance critical
- PER.4: Don't assume that complicated code is necessarily faster than simple code
- PER.5: Don't assume that low-level code is necessarily faster than high-level code
- PER.6: Don't make claims about performance without measurements
- PER.10: Rely on the static type system
- PER.11: Move computation from run time to compile time
- PER.12: Eliminate redundant aliases
- PER.13: Eliminate redundant indirections
- PER.14: Minimize the number of allocations and deallocations
- PER.15: Do not allocate on a critical branch
- PER.16: Use compact data structures
- PER.17: Declare the most used member of a time-critical struct first
- PER.18: Space is time
- PER.19: Access memory predictably
- PER.30: Avoid context switches on the critical path

PER.1: Don't optimize without reason

Reason If there is no need for optimization, the main result of the effort will be more errors and higher maintenance costs.

Note Some people optimize out of habit or because it's fun.

???

PER.2: Don't optimize prematurely

Reason Elaborately optimized code is usually larger and harder to change than unoptimized code.

???

PER.3: Don't optimize something that's not performance critical

Reason Optimizing a non-performance-critical part of a program has no effect on system performance.

Note If your program spends most of its time waiting for the web or for a human, optimization of in-memory computation is probably useless. ???

PER.4: Don't assume that complicated code is necessarily faster than simple code

Reason Simple code can be very fast. Optimizers sometimes do marvels with simple code

Example, good

```
// clear expression of intent, fast execution

vector<uint8_t> v(100000);

for(auto& c : v)
    c = ~c;
```

Example, bad

```
// intended to be faster, but is actually slower

vector<uint8_t> v(100000);

for(size_t i=0; i<v.size(); i+=sizeof(uint64_t))
{
    uint64_t& quad_word = *reinterpret_cast<uint64_t*>(&v[i]);
    quad_word = ~quad_word;
}
```

Note ???

???

PER.5: Don't assume that low-level code is necessarily faster than high-level code

Reason Low-level code sometimes inhibits optimizations. Optimizers sometimes do marvels with high-level code.

Note ???

???

PER.6: Don't make claims about performance without measurements

Reason The field of performance is littered with myth and bogus folklore. Modern hardware and optimizers defy naive assumptions; even experts are regularly surprised.

Note Getting good performance measurements can be hard and require specialized tools.

Note A few simple microbenchmarks using Unix `time` or the standard library `<chrono>` can help dispel the most obvious myths. If you can't measure your complete system accurately, at least try to measure a few of your key operations and algorithms. A profiler can help tell you which parts of your system are performance critical. Often, you will be surprised.

???

PER.10: Rely on the static type system

Reason Type violations, weak types (e.g. `void*`s), and low level code (e.g., manipulation of sequences as individual bytes) make the job of the optimizer much harder. Simple code often optimizes better than hand-crafted complex code.

???

PER.11: Move computation from run time to compile time

???

PER.12: Eliminate redundant aliases

???

PER.13: Eliminate redundant indirections

???

PER.14: Minimize the number of allocations and deallocations

???

PER.15: Do not allocate on a critical branch

???

PER.16: Use compact data structures

Reason Performance is typically dominated by memory access times.

???

PER.17: Declare the most used member of a time-critical struct first

???

PER.18: Space is time

Reason Performance is typically dominated by memory access times.

???

PER.19: Access memory predictably

Reason Performance is very sensitive to cache performance and cache algorithms favor simple (usually linear) access to adjacent data.

Example

```
int matrix[rows][cols];

// bad
for (int c = 0; c < cols; ++c)
    for (int r = 0; r < rows; ++r)
        sum += matrix[r][c];
```

```
// good
for (int r = 0; r < rows; ++r)
    for (int c = 0; c < cols; ++c)
        sum += matrix[r][c];
```

PER.30: Avoid context switches on the critical path

???

CP: Concurrency and Parallelism

???

Concurrency and parallelism rule summary:

- CP.1: Assume that your code will run as part of a multi-threaded program
- CP.2: Avoid data races

See also:

- CP.con: Concurrency
- CP.par: Parallelism
- CP.simd: SIMD
- CP.free: Lock-free programming

CP.1: Assume that your code will run as part of a multi-threaded program

Reason It is hard to be certain that concurrency isn't used now or sometime in the future. Code gets re-used. Libraries using threads may be used from some other part of the program. Note that this applies most urgently to library code and least urgently to stand-alone applications.

Example

```
double cached_computation(double x)
{
    static double cached_x = 0.0;
    static double cached_result = COMPUTATION_OF_ZERO;
    double result;

    if (cached_x = x)
```

```
        return cached_result;
    result = computation(x);
    cached_x = x;
    cached_result = result;
    return result;
}
```

Although `cached_computation` works perfectly in a single-threaded environment, in a multi-threaded environment the two `static` variables result in data races and thus undefined behavior.

There are several ways that this example could be made safe for a multi-threaded environment:

- Delegate concurrency concerns upwards to the caller.
- Mark the `static` variables as `thread_local` (which might make caching less effective).
- Implement concurrency control, for example, protecting the two `static` variables with a `static` lock (which might reduce performance).
- Have the caller provide the memory to be used for the cache, thereby delegating both memory allocation and concurrency concerns upwards to the caller.
- Refuse to build and/or run in a multi-threaded environment.
- Provide two implementations, one which is used in single-threaded environments and another which is used in multi-threaded environments.

Exception: There are examples where code will never be run in a multi-threaded environment. However, there are also many examples where code that was "known" to never run in a multi-threaded program was run as part of a multi-threaded program. Often years later. Typically, such programs lead to a painful effort to remove data races. Therefore, code that is never intended to run in a multi-threaded environment should be clearly labeled as such and ideally come with compile or run-time enforcement mechanisms to catch those usage bugs early.

CP.2: Avoid data races

Reason Unless you do, nothing is guaranteed to work and subtle errors will persist.

Note In a nutshell, if two threads can access the same named object concurrently (without synchronization), and at least one is a writer (performing a non-`const` operation), you have a data race. For further information of how to use synchronization well to eliminate data races, please consult a good book about concurrency.

Enforcement Some is possible, do at least something.

CP.con: Concurrency

???

Concurrency rule summary:

- ???
- ???

???? should there be a "use X rather than `std::async`" where X is something that would use a better specified thread pool?

Speaking of concurrency, should there be a note about the dangers of `std::atomic` (weapons)? A lot of people, myself included, like to experiment with `std::memory_order`, but it is perhaps best to keep a close watch on those things in production code. Even vendors mess this up: Microsoft had to fix their `shared_ptr` (weak refcount decrement wasn't synchronized-with the destructor, if I recall correctly, although it was only a problem on ARM, not Intel) and everyone (gcc, clang, Microsoft, and Intel) had to fix their `compare_exchange_*` this year, after an implementation bug caused losses to some finance company and they were kind enough to let the community know.

It should definitely be mentioned that `volatile` does not provide atomicity, does not synchronize between threads, and does not prevent instruction reordering (neither compiler nor hardware), and simply has nothing to do with concurrency.

```
if (source->pool != YARROW_FAST_POOL && source->pool != YARROW_SLOW_POOL) {
    THROW(YARROW_BAD_SOURCE);
}
```

??? Is `std::async` worth using in light of future (and even existing, as libraries) parallelism facilities? What should the guidelines recommend if someone wants to parallelize, e.g., `std::accumulate` (with the additional precondition of commutativity), or merge sort?

???UNIX signal handling???. May be worth reminding how little is async-signal-safe, and how to communicate with a signal handler (best is probably "not at all")

CP.par: Parallelism

???

Parallelism rule summary:

- ???
- ???

CP.simd: SIMD

???

SIMD rule summary:

- ???
- ???

CP.free: Lock-free programming

???

Lock-free programming rule summary:

- ???
- ???

Don't use lock-free programming unless you absolutely have to

Reason It's error-prone and requires expert level knowledge of language features, machine architecture, and data structures.

Alternative: Use lock-free data structures implemented by others as part of some library.

E: Error handling

Error handling involves:

- Detecting an error
- Transmitting information about an error to some handler code
- Preserve the state of a program in a valid state
- Avoid resource leaks

It is not possible to recover from all errors. If recovery from an error is not possible, it is important to quickly "get out" in a well-defined way. A strategy for error handling must be simple, or it becomes a source of even worse errors. Untested and rarely executed error-handling code is itself the source of many bugs.

The rules are designed to help avoid several kinds of errors:

- Type violations (e.g., misuse of `union`s and casts)
- Resource leaks (including memory leaks)

- Bounds errors
- Lifetime errors (e.g., accessing an object after is has been `delete`d)
- Complexity errors (logical errors make likely by overly complex expression of ideas)
- Interface errors (e.g., an unexpected value is passed through an interface)

Error-handling rule summary:

- E.1: Develop an error-handling strategy early in a design
- E.2: Throw an exception to signal that a function can't perform its assigned task
- E.3: Use exceptions for error handling only
- E.4: Design your error-handling strategy around invariants
- E.5: Let a constructor establish an invariant, and throw if it cannot
- E.6: Use RAII to prevent leaks
- E.7: State your preconditions
- E.8: State your postconditions
- E.12: Use `noexcept` when exiting a function because of a `throw` is impossible or unacceptable
- E.13: Never throw while being the direct owner of an object
- E.14: Use purpose-designed user-defined types as exceptions (not built-in types)
- E.15: Catch exceptions from a hierarchy by reference
- E.16: Destructors, deallocation, and `swap` must never fail
- E.17: Don't try to catch every exception in every function
- E.18: Minimize the use of explicit `try`/`catch`
- E.19: Use a `final_action` object to express cleanup if no suitable resource handle is available
- E.25: If you can't throw exceptions, simulate RAII for resource management
- E.26: If you can't throw exceptions, consider failing fast
- E.27: If you can't throw exceptions, use error codes systematically
- E.28: Avoid error handling based on global state (e.g. `errno`)

E.1: Develop an error-handling strategy early in a design

Reason A consistent and complete strategy for handling errors and resource leaks is hard to retrofit into a system.

E.2: Throw an exception to signal that a function can't perform its assigned task

Reason To make error handling systematic, robust, and non-repetitive.

Example

```
struct Foo {
    vector<Thing> v;
    File_handle f;
    string s;
};

void use()
{
    Foo bar { {Thing{1}, Thing{2}, Thing{monkey}}, {"my_file", "r"}, "Here we go!"};
    // ...
}
```

Here, `vector` and `strings` constructors may not be able to allocate sufficient memory for their elements, `vectors` constructor may not be able copy the `Things` in its initializer list, and `File_handle` may not be able to open the required file. In each case, they throw an exception for `use()`'s caller to handle. If `use()` could handle the failure to construct `bar` it can take control using `try`/`catch`. In either case, `Foo`'s constructor correctly destroys constructed members before passing control to whatever tried to create a `Foo`. Note that there is no return value that could contain an error code.

The `File_handle` constructor might defined like this:

```
File_handle::File_handle(const string& name, const string& mode)
    :f{fopen(name.c_str(), mode.c_str())}
{
    if (!f)
        throw runtime_error{"File_handle: could not open "S-+ name + " as " + mode"}
}
```

Note It is often said that exceptions are meant to signal exceptional events and failures. However, that's a bit circular because "what is exceptional?" Examples:

- A precondition that cannot be met

- A constructor that cannot construct an object (failure to establish its class's invariant)
- An out-of-range error (e.g., `v[v.size()] =7`)
- Inability to acquire a resource (e.g., the network is down)

In contrast, termination of an ordinary loop is not exceptional. Unless the loop was meant to be infinite, termination is normal and expected.

Note Don't use a `throw` as simply an alternative way of returning a value from a function.

Exception: Some systems, such as hard-real time systems require a guarantee that an action is taken in a (typically short) constant maximum time known before execution starts. Such systems can use exceptions only if there is tool support for accurately predicting the maximum time to recover from a `throw`.

See also: RAII

See also: discussion

Note Before deciding that you cannot afford or don't like exception-based error handling, have a look at the alternatives.

E.3: Use exceptions for error handling only

Reason To keep error handling separated from "ordinary code." C++ implementations tend to be optimized based on the assumption that exceptions are rare.

Example, don't

```
int find_index(vector<string>& vec, const string& x)   // don't: exception not used for
{
    try {
        for (int i =0; i < vec.size(); ++i)
            if (vec[i] == x) throw i;  // found x
    } catch (int i) {
        return i;
    }
    return -1;   // not found
}
```

This is more complicated and most likely runs much slower than the obvious alternative. There is nothing exceptional about finding a value in a `vector`.

E.4: Design your error-handling strategy around invariants

Reason To use an object it must be in a valid state (defined formally or informally by an invariant) and to recover from an error every object not destroyed must be in a valid state.

Note An invariant is logical condition for the members of an object that a constructor must establish for the public member functions to assume.

E.5: Let a constructor establish an invariant, and throw if it cannot

Reason Leaving an object without its invariant established is asking for trouble. Not all member functions can be called.

Example

```
???
```

See also: If a constructor cannot construct a valid object, throw an exception

Enforcement ???

E.6: Use RAII to prevent leaks

Reason Leaks are typically unacceptable. RAII ("Resource Acquisition Is Initialization") is the simplest, most systematic way of preventing leaks.

Example

```
void f1(int i)   // Bad: possibly leak
{
    int* p = new int[12];
    // ...
    if (i < 17) throw Bad {"in f()", i};
    // ...
}
```

We could carefully release the resource before the throw:

```
void f2(int i)   // Clumsy: explicit release
{
    int* p = new int[12];
```

```
    // ...
    if (i < 17) {
        delete p;
        throw Bad {"in f()", i};
    }
    // ...
}
```

This is verbose. In larger code with multiple possible `throw`s explicit releases become repetitive and error-prone.

```
void f3(int i)   // OK: resource management done by a handle
{
    auto p = make_unique<int[]>(12);
    // ...
    if (i < 17) throw Bad {"in f()", i};
    // ...
}
```

Note that this works even when the `throw` is implicit because it happened in a called function:

```
void f4(int i)   // OK: resource management done by a handle
{
    auto p = make_unique<int[]>(12);
    // ...
    helper(i);   // may throw
    // ...
}
```

Unless you really need pointer semantics, use a local resource object:

```
void f5(int i)   // OK: resource management done by local object
{
    vector<int> v(12);
    // ...
    helper(i);   // may throw
    // ...
}
```

Note If there is no obvious resource handle, cleanup actions can be represented by a `final_action` object

Note But what do we do if we are writing a program where exceptions cannot be used? First challenge that assumption; there are many anti-exceptions myths around. We know of only a few good reasons:

- We are on a system so small that the exception support would eat up most of our 2K or memory.
- We are in a hard-real-time system and we don't have tools that guarantee us that an exception is handled within the required time.
- We are in a system with tons of legacy code using lots of pointers in difficult-to-understand ways (in particular without a recognizable ownership strategy) so that exceptions could cause leaks.
- We get fired if we challenge our manager's ancient wisdom.

Only the first of these reasons is fundamental, so whenever possible, use exceptions to implement RAII, or design your RAII objects to never fail. When exceptions cannot be used, simulate RAII. That is, systematically check that objects are valid after construction and still release all resources in the destructor. One strategy is to add a `valid()` operation to every resource handle:

```
void f()
{
    vector<string> vs(100);   // not std::vector: valid() added
    if (!vs.valid()) {
        // handle error or exit
    }

    Ifstream fs("foo");   // not std::ifstream: valid() added
    if (!fs.valid()) {
        // handle error or exit
    }

    // ...
} // destructors clean up as usual
```

Obviously, this increases the size of the code, doesn't allow for implicit propagation of "exceptions" (`valid()` checks), and `valid()` checks can be forgotten. Prefer to use exceptions.

See also: discussion.

Enforcement ???

E.7: State your preconditions

Reason To avoid interface errors.

See also: precondition rule.

E.8: State your postconditions

Reason To avoid interface errors.

See also: postcondition rule.

E.12: Use `noexcept` when exiting a function because of a `throw` is impossible or unacceptable

Reason To make error handling systematic, robust, and efficient.

Example

```
double compute(double d) noexcept
{
    return log(sqrt(d <= 0 ? 1 : d));
}
```

Here, I know that `compute` will not throw because it is composed out of operations that don't throw. By declaring `compute` to be `noexcept` I give the compiler and human readers information that can make it easier for them to understand and manipulate `compute`.

Note Many standard library functions are `noexcept` including all the standard library functions "inherited" from the C standard library.

Example

```
vector<double> munge(const vector<double>& v) noexcept
{
    vector<double> v2(v.size());
    // ... do something ...
}
```

The `noexcept` here states that I am not willing or able to handle the situation where I cannot construct the local `vector`. That is, I consider memory exhaustion a serious design error (on par with hardware failures) so that I'm willing to crash the program if it happens.

See also: discussion.

E.13: Never throw while being the direct owner of an object

Reason That would be a leak.

Example

```
void leak(int x)   // don't: may leak
{
    auto p = new int{7};
    if (x < 0) throw Get_me_out_of_here{}  // may leak *p
    // ...
    delete p;   // we may never get here
}
```

One way of avoiding such problems is to use resource handles consistently:

```
void no_leak(int x)
{
    auto p = make_unique<int>(7);
    if (x < 0) throw Get_me_out_of_here{};  // will delete *p if necessary
    // ...
    // no need for delete p
}
```

See also: ???resource rule ???

E.14: Use purpose-designed user-defined types as exceptions (not built-in types)

Reason A user-defined type is unlikely to clash with other people's exceptions.

Example

```
void my_code()
{
    // ...
    throw Moonphase_error{};
    // ...
}

void your_code()
{
    try {
        // ...
        my_code();
        // ...
    }
    catch(Bufferpool_exhausted) {
        // ...
    }
}
```

Example, don't

```
void my_code()     // Don't
{
    // ...
    throw 7;       // 7 means "moon in the 4th quarter"
    // ...
}

void your_code()   // Don't
{
    try {
        // ...
        my_code();
        // ...
    }
    catch(int i) {  // i == 7 means "input buffer too small"
        // ...
    }
}
```

Note The standard-library classes derived from `exception` should be used only as base classes or for exceptions that require only "generic" handling. Like built-in types, their use could clash with other people's use of them.

Example, don't

```
void my_code()   // Don't
{
    // ...
    throw runtime_error{"moon in the 4th quarter"};
    // ...
}

void your_code()   // Don't
{
    try {
        // ...
        my_code();
        // ...
    }
    catch(runtime_error) {   // runtime_error means "input buffer too small"
        // ...
```

```
    }
}
```

See also: Discussion

Enforcement Catch `throw` and `catch` of a built-in type. Maybe warn about `throw` and `catch` using an standard-library `exception` type. Obviously, exceptions derived from the `std::exception` hierarchy is fine.

E.15: Catch exceptions from a hierarchy by reference

Reason To prevent slicing.

Example

```
void f()
try {
    // ...
}
catch (exception e) {   // don't: may slice
    // ...
}
```

Instead, use:

```
catch (exception& e) { /* ... */ }
```

Enforcement Flag by-value exceptions if their types are part of a hierarchy (could require whole-program analysis to be perfect).

E.16: Destructors, deallocation, and `swap` must never fail

Reason We don't know how to write reliable programs if a destructor, a swap, or a memory deallocation fails; that is, if it exits by an exception or simply doesn't perform its required action.

Example, don't

```
class Connection {
    // ...
public:
    ~Connection()   // Don't: very bad destructor
    {
        if (cannot_disconnect()) throw I_give_up{information};
        // ...
    }
};
```

Note Many have tried to write reliable code violating this rule for examples such as a network connection that "refuses to close". To the best of our knowledge nobody has found a general way of doing this though occasionally, for very specific examples, you can get away with setting some state for future cleanup. Every example we have seen of this is error-prone, specialized, and usually buggy.

Note The standard library assumes that destructors, deallocation functions (e.g., `operator delete`), and `swap` do not throw. If they do, basic standard library invariants are broken.

Note Deallocation functions, including `operator delete`, must be `noexcept`. `swap` functions must be `noexcept`. Most destructors are implicitly `noexcept` by default.

Enforcement Catch destructors, deallocation operations, and `swap`s that `throw`. Catch such operations that are not `noexcept`.

See also: discussion

E.17: Don't try to catch every exception in every function

Reason Catching an exception in a function that cannot take a meaningful recovery action leads to complexity and waste. Let an exception propagate until it reaches a function that can handle it. Let cleanup actions on the unwinding path be handled by RAII.

Example, don't

```
void f()   // bad
{
    try {
        // ...
```

```
    }
    catch (...) {
        throw;    // propagate exception
    }
}
```

Enforcement

- Flag nested try-blocks.
- Flag source code files with a too high ratio of try-blocks to functions. (??? Problem: define "too high")

E.18: Minimize the use of explicit `try`/`catch`

Reason `try`/`catch` is verbose and non-trivial uses error-prone. `try`/`catch` can be a sign of unsystematic and/or low-level resource management or error handling.

Example, Bad

```
void f(zstring s)
{
    Gadget* p;
    try {
        p = new Gadget(s);
        // ...
    }
    catch (Gadget_construction_failure) {
        delete p;
        throw;
    }
}
```

This code is messy. There could be a leak from the naked pointer in the `try` block. Not all exceptiones are handled. `deleting` an object that failed to construct is almost certainly a mistake. Better:

```
void f2(zstring s)
{
    Gadget g {s};
}
```

Alternatives

- proper resource handles and RAII
- `finally`

Enforcement ??? hard, needs a heuristic

E.19: Use a `final_action` object to express cleanup if no suitable resource handle is available

Reason `finally` is less verbose and harder to get wrong than `try`/`catch`.

Example

```
void f(int n)
{
    void* p = malloc(1, n);
    auto _ = finally([p] { free(p); });
    // ...
}
```

Note `finally` is not as messy as `try`/`catch`, but it is still ad-hoc. Prefer proper resource management objects.

E.25: If you can't throw exceptions, simulate RAII for resource management

Reason Even without exceptions, RAII is usually the best and most systematic way of dealing with resources.

Note Error handling using exceptions is the only complete and systematic way of handling non-local errors in C++. In particular, non-intrusively signalling failure to construct an object requires an exception. Signalling errors in a way that cannot be ignored requires exceptions. If you can't use exceptions, simulate their use as best you can.

A lot of fear of exceptions is misguided. When used for exceptional circumstances in code that is not littered with pointers and complicated control structures, exception handling is almost always affordable (in time and space) and almost always leads to better code. This, of course, assumes a good implementation of the exception handling mechanisms, which is not available on all systems. There are also cases where the problems above do not apply, but exceptions cannot be used for other reasons. Some hard real-time systems are an example: An operation has to be completed within a fixed time with an error or a correct answer. In the absence of appropriate time estimation tools, this is hard to guarantee for exceptions. Such systems (e.g. flight control software) typically also ban the use of dynamic (heap) memory.

So, the primary guideline for error handling is "use exceptions and RAII." This section deals with the cases where you either do not have an efficient implementation or exceptions or have such a rat's nest of old-style code (e.g., lots of pointers, ill-defined ownership, and lots of unsystematic error handling based on tests of errors codes) that it is infeasible to introduce simple and systematic exception handling.

Before condemning exceptions or complaining too much about their cost, consider examples of the use of error codes.

Example Assume you wanted to write

```
void func(int n)
{
    Gadget g(n);
    // ...
}
```

If the `gadget` isn't correctly constructed, `func` exits with an exception. If we cannot throw an exception, we can simulate this RAII style of resource handling by adding a `valid()` member function to `Gadget`:

```
error_indicator func(int n)
{
    Gadget g(n);
    if (!g.valid()) return gadget_construction_error;
    // ...
    return 0;   // zero indicates "good"
}
```

The problem is of course that the caller now have to remember to test the return value.

See also: Discussion.

Enforcement Possible (only) for specific versions of this idea: e.g., test for systematic test of `valid()` after resource handle construction

E.26: If you can't throw exceptions, consider failing fast

Reason If you can't do a good job at recovering, at least you can get out before too much consequential damage is done.

See also Simulating RAII.

Note If you cannot be systematic about error handling, consider "crashing" as a response to any error that cannot be handled locally. That is, if you cannot recover from an error in the context of the function that detected it, call `abort()`, `quick_exit()`, or a similar function that will trigger some sort of system restart.

In systems where you have lots of processes and/or lots of computers, you need to expect and handle fatal crashes anyway, say from hardware failures. In such cases, "crashing" is simply leaving error handling to the next level of the system.

Example

```
void do_something(int n)
{
       // ...
       p = static_cast<X*>(malloc(n,X));
       if (p==nullptr) abort();     // abort if memory is exhausted
       // ...
 }
```

Most systems cannot handle memory exhaustion gracefully anyway. This is roughly equivalent to

```
void do_something(Int n)
{
       // ...
       p = new X[n];    // throw if memory is exhausted (by default, terminate)
       // ...
 }
```

Typically, it is a good idea to log the reason for the "crash" before exiting.

Enforcement Awkward

E.27: If you can't throw exceptions, use error codes systematically

Reason Systematic use of any error-handling strategy minimizes the chance of forgetting to handle an error.

See also Simulating RAII.

Note There are several issues to be addressed:

- how do you transmit an error indicator from out of a function?
- how do you release all resources from a function before doing an error exit?
- What do you use as an error indicator?

In general, returning an error indicator implies returning two values: The result and an error indicator. The error indicator can be part of the object, e.g. an object can have a `valid()` indicator or a pair of values can be returned.

Example

```
Gadget make_gadget(int n)
{
    // ...
}

void user()
{
    Gadget g = make_gadget(17);
    if (!g.valid()) {
            // error handling
    }
    // ...
}
```

This approach fits with simulated RAII resource management. The `valid()` function could return an `error_indicator` (e.g. a member of an `error_indicator` enumeration).

Example What if we cannot or do not want to modify the `Gadget` type? In that case, we must return a pair of values. For example:

```
std::pair<Gadget,error_indicator> make_gadget(int n)
{
    // ...
}

void user()
{
    auto r = make_gadget(17);
    if (!r.second) {
            // error handling
    }
```

```
    Gadget& g = r.first;
    // ...
}
```

As shown, `std::pair` is a possible return type. Some people prefer a specific type. For example:

```
Gval make_gadget(int n)
{
    // ...
}

void user()
{
    auto r = make_gadget(17);
    if (!r.err) {
            // error handling
    }
    Gadget& g = r.val;
    // ...
}
```

One reason to prefer a specific return type is to have names for its members, rather than the somewhat cryptic `first` and `second` and to avoid confusion with other uses of `std::pair`.

Example

In general, you must clean up before an eror exit. This can be messy:

```
std::pair<int,error_indicator> user()
{
    Gadget g1 = make_gadget(17);
    if (!g1.valid()) {
            return {0,g1_error};
    }

    Gadget g2 = make_gadget(17);
    if (!g2.valid()) {
            cleanup(g1);
            return {0,g2_error};
    }

    // ...

    if (all_foobar(g1,g2)) {
```

```
        cleanup(g1);
        cleanup(g2);
        return {0,foobar_error};
    // ...

    cleanup(g1);
    cleanup(g2);
    return {res,0};
}
```

Simulating RAII can be non-trivial, especially in functions with multiple resources and multiple possible errors. A not uncommon technique is to gather cleanup at the end of the function to avoid repetittion:

```
std::pair<int,error_indicator> user()
{
    error_indicator err = 0;

    Gadget g1 = make_gadget(17);
    if (!g1.valid()) {
            err = g2_error;
            goto exit;
    }

    Gadget g2 = make_gadget(17);
    if (!g2.valid()) {
            err = g2_error;
            goto exit;
    }

    if (all_foobar(g1,g2)) {
        err = foobar_error;
        goto exit;
    }
    // ...
```

exit:
if (g1.valid()) cleanup(g1); if (g1.valid()) cleanup(g2); return {res,err}; }

The larger the function, the more tempting this technique becomes. Aso, the larger the program becomes the harder it is to apply an error-indicator-based error handling strategy systematically.

We prefer exception-based error handling and recommend keeping functions short.

See also: Discussion.

Enforcement Awkward.

E.28: Avoid error handling based on global state (e.g. `errno`)

Reason Global state is hard to manage and it is easy to forget to check it. When did you last test the return value of `printf()`?

See also Simulating RAII.

Example, bad

```
???
```

Note C-stye error handling is based on the global variable `errno`, so it is essentially impossible to avoid this style completely.

Enforcement Awkward.

Con: Constants and Immutability

You can't have a race condition on a constant. It is easier to reason about a program when many of the objects cannot change their values. Interfaces that promises "no change" of objects passed as arguments greatly increase readability.

Constant rule summary:

- Con.1: By default, make objects immutable
- Con.2: By default, make member functions `const`
- Con.3: By default, pass pointers and references to `const`s
- Con.4: Use `const` to define objects with values that do not change after construction
- Con.5: Use `constexpr` for values that can be computed at compile time

Con.1: By default, make objects immutable

Reason Immutable objects are easier to reason about, so make object non-`const` only when there is a need to change their value. Prevents accidental or hard-to-notice change of value.

Example

```
for (const string& s : c) cout << s << '\n';    // just reading: const

for (string& s : c) cout << s << '\n';    // BAD: just reading

for (string& s: c) cin>>s;  // needs to write: non-const
```

Exception Function arguments are rarely mutated, but also rarely declared const. To avoid confusion and lots of false positives, don't enforce this rule for function arguments.

```
void f(const char*const p); // pedantic
void g(const int i);        // pedantic
```

Note that function parameter is a local variable so changes to it are local.

Enforcement

- Flag non-const variables that are not modified (except for parameters to avoid many false positives)

Con.2: By default, make member functions `const`

Reason A member function should be marked `const` unless it changes the object's observable state. This gives a more precise statement of design intent, better readability, more errors caught by the compiler, and sometimes more optimization opportunities.

Example; bad

```
class Point {
    int x, y;
public:
    int getx() { return x; }    // BAD, should be const as it doesn't modify the objec
    // ...
};

void f(const Point& pt) {
    int x = pt.getx();          // ERROR, doesn't compile because getx was not marked c
}
```

Note Do not cast away `const`.

Enforcement

- Flag a member function that is not marked `const`, but that does not perform a non-`const` operation on any member variable.

Con.3: By default, pass pointers and references to `const`s

Reason To avoid a called function unexpectedly changing the value. It's far easier to reason about programs when called functions don't modify state.

Example

```
void f(char* p);        // does f modify *p? (assume it does)
void g(const char* p);  // g does not modify *p
```

Note It is not inherently bad to pass a pointer or reference to non-const, but that should be done only when the called function is supposed to modify the object.

Note Do not cast away `const`.

Enforcement

- flag function that does not modify an object passed by pointer or reference to non-cost
- flag a function that (using a cast) modifies an object passed by pointer or reference to const

Con.4: Use `const` to define objects with values that do not change after construction

Reason Prevent surprises from unexpectedly changed object values.

Example

```
void f()
{
    int x = 7;
    const int y = 9;

    for (;;) {
        // ...
    }
    // ...
}
```

As `x` is not const, we must assume that it is modified somewhere in the loop.

Enforcement

- Flag unmodified non-const variables.

Con.5: Use `constexpr` for values that can be computed at compile time

Reason Better performance, better compile-time checking, guaranteed compile-time evaluation, no possibility of race conditions.

Example

```
double x = f(2);            // possible run-time evaluation
const double x = f(2);      // possible run-time evaluation
constexpr double y = f(2);  // error unless f(2) can be evaluated at compile time
```

Note See F.4.

Enforcement

- Flag `const` definitions with constant expression initializers.

T: Templates and generic programming

Generic programming is programming using types and algorithms parameterized by types, values, and algorithms. In C++, generic programming is supported by the `template` language mechanisms.

Arguments to generic functions are characterized by sets of requirements on the argument types and values involved. In C++, these requirements are expressed by compile-time predicates called concepts.

Templates can also be used for meta-programming; that is, programs that compose code at compile time.

Template use rule summary:

- T.1: Use templates to raise the level of abstraction of code
- T.2: Use templates to express algorithms that apply to many argument types
- T.3: Use templates to express containers and ranges
- T.4: Use templates to express syntax tree manipulation

- T.5: Combine generic and OO techniques to amplify their strengths, not their costs

Concept use rule summary:

- T.10: Specify concepts for all template arguments
- T.11: Whenever possible use standard concepts
- T.12: Prefer concept names over `auto` for local variables
- T.13: Prefer the shorthand notation for simple, single-type argument concepts
- ???

Concept definition rule summary:

- T.20: Avoid "concepts" without meaningful semantics
- T.21: Define concepts to define complete sets of operations
- T.22: Specify axioms for concepts
- T.23: Differentiate a refined concept from its more general case by adding new use patterns
- T.24: Use tag classes or traits to differentiate concepts that differ only in semantics
- T.25: Avoid negating constraints
- T.26: Prefer to define concepts in terms of use-patterns rather than simple syntax
- ???

Template interface rule summary:

- T.40: Use function objects to pass operations to algorithms
- T.41: Require complete sets of operations for a concept
- T.42: Use template aliases to simplify notation and hide implementation details
- T.43: Prefer `using` over `typedef` for defining aliases
- T.44: Use function templates to deduce class template argument types (where feasible)
- T.46: Require template arguments to be at least `Regular` or `SemiRegular`
- T.47: Avoid highly visible unconstrained templates with common names
- T.48: If your compiler does not support concepts, fake them with `enable_if`
- T.49: Where possible, avoid type-erasure
- T.50: Avoid writing an unconstrained template in the same namespace as a type

Template definition rule summary:

- T.60: Minimize a template's context dependencies
- T.61: Do not over-parameterize members (SCARY)
- T.62: Place non-dependent template members in a non-templated base class
- T.64: Use specialization to provide alternative implementations of class templates

- T.65: Use tag dispatch to provide alternative implementations of functions
- T.66: Use selection using `enable_if` to optionally define a function
- T.67: Use specialization to provide alternative implementations for irregular types
- T.68: Use `{}` rather than `()` within templates to avoid ambiguities
- T.69: Inside a template, don't make an unqualified nonmember function call unless you intend it to be a customization point

Template and hierarchy rule summary:

- T.80: Do not naively templatize a class hierarchy
- T.81: Do not mix hierarchies and arrays // ??? somewhere in "hierarchies"
- T.82: Linearize a hierarchy when virtual functions are undesirable
- T.83: Do not declare a member function template virtual
- T.84: Use a non-template core implementation to provide an ABI-stable interface
- T.??: ????

Variadic template rule summary:

- T.100: Use variadic templates when you need a function that takes a variable number of arguments of a variety of types
- T.101: ??? How to pass arguments to a variadic template ???
- T.102: ??? How to process arguments to a variadic template ???
- T.103: Don't use variadic templates for homogeneous argument lists
- T.??: ????

Metaprogramming rule summary:

- T.120: Use template metaprogramming only when you really need to
- T.121: Use template metaprogramming primarily to emulate concepts
- T.122: Use templates (usually template aliases) to compute types at compile time
- T.123: Use `constexpr` functions to compute values at compile time
- T.124: Prefer to use standard-library TMP facilities
- T.125: If you need to go beyond the standard-library TMP facilities, use an existing library
- T.??: ????

Other template rules summary:

- T.140: Name all nontrivial operations
- T.141: Use an unnamed lambda if you need a simple function object in one place only
- T.142: Use template variables to simplify notation
- T.143: Don't write unintentionally nongeneric code
- T.144: Don't specialize function templates
- T.??: ????

T.gp: Generic programming

Generic programming is programming using types and algorithms parameterized by types, values, and algorithms.

T.1: Use templates to raise the level of abstraction of code

Reason Generality. Re-use. Efficiency. Encourages consistent definition of user types.

Example, bad Conceptually, the following requirements are wrong because what we want of `T` is more than just the very low-level concepts of "can be incremented" or "can be added":

```
template<typename T, typename A>
    // requires Incrementable<T>
A sum1(vector<T>& v, A s)
{
    for (auto x : v) s+=x;
    return s;
}

template<typename T, typename A>
    // requires Simple_number<T>
A sum2(vector<T>& v, A s)
{
    for (auto x : v) s = s + x;
    return s;
}
```

Assuming that `Incrementable` does not support `+` and `Simple_number` does not support `+=`, we have overconstrained implementers of `sum1` and `sum2`. And, in this case, missed an opportunity for a generalization.

Example

```
template<typename T, typename A>
    // requires Arithmetic<T>
A sum(vector<T>& v, A s)
{
    for (auto x : v) s+=x;
    return s;
}
```

Assuming that `Arithmetic` requires both `+` and `+=`, we have constrained the user of `sum` to provide a complete arithmetic type. That is not a minimal requirement, but it gives the implementer of algorithms much needed freedom and ensures that any `Arithmetic` type can be used for a wide variety of algorithms.

For additional generality and reusability, we could also use a more general `Container` or `Range` concept instead of committing to only one container, `vector`.

Note If we define a template to require exactly the operations required for a single implementation of a single algorithm (e.g., requiring just `+=` rather than also `=` and `+`) and only those, we have overconstrained maintainers. We aim to minimize requirements on template arguments, but the absolutely minimal requirements of an implementation is rarely a meaningful concept.

Note Templates can be used to express essentially everything (they are Turing complete), but the aim of generic programming (as expressed using templates) is to efficiently generalize operations/algorithms over a set of types with similar semantic properties.

Enforcement

- Flag algorithms with "overly simple" requirements, such as direct use of specific operators without a concept.
- Do not flag the definition of the "overly simple" concepts themselves; they may simply be building blocks for more useful concepts.

T.2: Use templates to express algorithms that apply to many argument types

Reason Generality. Minimizing the amount of source code. Interoperability. Re-use.

Example That's the foundation of the STL. A single `find` algorithm easily works with any kind of input range:

```
template<typename Iter, typename Val>
    // requires Input_iterator<Iter>
    //       && Equality_comparable<Value_type<Iter>, Val>
Iter find(Iter b, Iter e, Val v)
{
    // ...
}
```

Note Don't use a template unless you have a realistic need for more than one template argument type. Don't overabstract.

Enforcement ??? tough, probably needs a human

T.3: Use templates to express containers and ranges

Reason Containers need an element type, and expressing that as a template argument is general, reusable, and type safe. It also avoids brittle or inefficient workarounds. Convention: That's the way the STL does it.

Example

```
template<typename T>
    // requires Regular<T>
class Vector {
    // ...
    T* elem;   // points to sz Ts
    int sz;
};

vector<double> v(10);
v[7] = 9.9;
```

Example, bad

```
class Container {
    // ...
    void* elem;   // points to size elements of some type
    int sz;
};

Container c(10, sizeof(double));
((double*)c.elem)[] = 9.9;
```

This doesn't directly express the intent of the programmer and hides the structure of the program from the type system and optimizer.

Hiding the `void*` behind macros simply obscures the problems and introduces new opportunities for confusion.

Exceptions: If you need an ABI-stable interface, you might have to provide a base implementation and express the (type-safe) template in terms of that. See Stable base.

Enforcement

- Flag uses of `void*`s and casts outside low-level implementation code

T.4: Use templates to express syntax tree manipulation

Reason ???

Example

```
???
```

Exceptions: ???

T.5: Combine generic and OO techniques to amplify their strengths, not their costs

Reason Generic and OO techniques are complementary.

Example Static helps dynamic: Use static polymorphism to implement dynamically polymorphic interfaces.

```
class Command {
    // pure virtual functions
};

// implementations
template</*...*/>
class ConcreteCommand : public Command {
    // implement virtuals
};
```

Example Dynamic helps static: Offer a generic, comfortable, statically bound interface, but internally dispatch dynamically, so you offer a uniform object layout. Examples include type erasure as with `std::shared_ptr`'s deleter. (But don't overuse type erasure.)

Note In a class template, nonvirtual functions are only instantiated if they're used – but virtual functions are instantiated every time. This can bloat code size, and may overconstrain a generic type by instantiating functionality that is never needed. Avoid this, even though the standard facets made this mistake.

Enforcement

- Flag a class template that declares new (non-inherited) virtual functions.

T.concepts: Concept rules

Concepts is a facility for specifying requirements for template arguments. It is an ISO technical specification, but not yet supported by currently shipping compilers. Concepts are, however, crucial in the thinking about generic programming and the basis of much work on future C++ libraries (standard and other).

Concept use rule summary:

- T.10: Specify concepts for all template arguments
- T.11: Whenever possible use standard concepts
- T.12: Prefer concept names over `auto`
- T.13: Prefer the shorthand notation for simple, single-type argument concepts
- ???

Concept definition rule summary:

- T.20: Avoid "concepts" without meaningful semantics
- T.21: Define concepts to define complete sets of operations
- T.22: Specify axioms for concepts
- T.23: Differentiate a refined concept from its more general case by adding new use patterns
- T.24: Use tag classes or traits to differentiate concepts that differ only in semantics
- T.25: Avoid negating constraints
- T.26: Prefer to define concepts in terms of use-patterns rather than simple syntax
- ???

T.con-use: Concept use

T.10: Specify concepts for all template arguments

Reason Correctness and readability. The assumed meaning (syntax and semantics) of a template argument is fundamental to the interface of a template. A concept dramatically improves documentation and error handling for the template. Specifying concepts for template arguments is a powerful design tool.

Example

```
template<typename Iter, typename Val>
    requires Input_iterator<Iter>
             && Equality_comparable<Value_type<Iter>, Val>
Iter find(Iter b, Iter e, Val v)
{
```

```
    // ...
}
```

or equivalently and more succinctly:

```
template<Input_iterator Iter, typename Val>
    requires Equality_comparable<Value_type<Iter>, Val>
Iter find(Iter b, Iter e, Val v)
{
    // ...
}
```

Note Until your compilers support the concepts language feature, leave the concepts in comments:

```
template<typename Iter, typename Val>
    // requires Input_iterator<Iter>
    //       && Equality_comparable<Value_type<Iter>, Val>
Iter find(Iter b, Iter e, Val v)
{
    // ...
}
```

Note Plain `typename` (or `auto`) is the least constraining concept. It should be used only rarely when nothing more than "it's a type" can be assumed. This is typically only needed when (as part of template metaprogramming code) we manipulate pure expression trees, postponing type checking.

References: TC++PL4, Palo Alto TR, Sutton

Enforcement Flag template type arguments without concepts

T.11: Whenever possible use standard concepts

Reason "Standard" concepts (as provided by the GSL, the ISO concepts TS, and hopefully soon the ISO standard itself) saves us the work of thinking up our own concepts, are better thought out than we can manage to do in a hurry, and improves interoperability.

Note Unless you are creating a new generic library, most of the concepts you need will already be defined by the standard library.

Example

```
concept<typename T>
// don't define this: Sortable is in the GSL
Ordered_container = Sequence<T> && Random_access<Iterator<T>> && Ordered<Value_type<T>>;

void sort(Ordered_container& s);
```

This `Ordered_container` is quite plausible, but it is very similar to the `Sortable` concept in the GSL (and the Range TS). Is it better? Is it right? Does it accurately reflect the standard's requirements for `sort`? It is better and simpler just to use `Sortable`:

```
void sort(Sortable& s);    // better
```

Note The set of "standard" concepts is evolving as we approach real (ISO) standardization.

Note Designing a useful concept is challenging.

Enforcement Hard.

- Look for unconstrained arguments, templates that use "unusual"/non-standard concepts, templates that use "homebrew" concepts without axioms.
- Develop a concept-discovery tool (e.g., see an early experiment).

T.12: Prefer concept names over `auto` for local variables

Reason `auto` is the weakest concept. Concept names convey more meaning than just `auto`.

Example

```
vector<string> v;
auto& x = v.front();      // bad
String& s = v.begin();    // good
```

Enforcement

- ???

T.13: Prefer the shorthand notation for simple, single-type argument concepts

Reason Readability. Direct expression of an idea.

Example To say "`T` is `Sortable`":

```
template<typename T>       // Correct but verbose: "The parameter is
    requires Sortable<T>   // of type T which is the name of a type
void sort(T&);             // that is Sortable"

template<Sortable T>       // Better: "The parameter is of type T
void sort(T&);             // which is Sortable"

void sort(Sortable&);      // Best: "The parameter is Sortable"
```

The shorter versions better match the way we speak. Note that many templates don't need to use the `template` keyword.

Enforcement

- Not feasible in the short term when people convert from the `<typename T>` and `<class T>` notation.
- Later, flag declarations that first introduces a typename and then constrains it with a simple, single-type-argument concept.

T.concepts.def: Concept definition rules

???

T.20: Avoid "concepts" without meaningful semantics

Reason Concepts are meant to express semantic notions, such as "a number", "a range" of elements, and "totally ordered." Simple constraints, such as "has a `+` operator" and "has a `>` operator" cannot be meaningfully specified in isolation and should be used only as building blocks for meaningful concepts, rather than in user code.

Example, bad

```
template<typename T>
concept Addable = has_plus<T>;    // bad; insufficient
```

```
template<Addable N> auto algo(const N& a, const N& b) // use two numbers
{
    // ...
    return a + b;
}

int x = 7;
int y = 9;
auto z = plus(x, y);   // z = 16

string xx = "7";
string yy = "9";
auto zz = plus(xx, yy);   // zz = "79"
```

Maybe the concatenation was expected. More likely, it was an accident. Defining minus equivalently would give dramatically different sets of accepted types. This `Addable` violates the mathematical rule that addition is supposed to be commutative: `a + b == b + a`.

Note The ability to specify a meaningful semantics is a defining characteristic of a true concept, as opposed to a syntactic constraint.

Example (using TS concepts)

```
template<typename T>
// The operators +, -, *, and / for a number are assumed to follow the usual mathemati
concept Number = has_plus<T>
                 && has_minus<T>
                 && has_multiply<T>
                 && has_divide<T>;

template<Number N> auto algo(const N& a, const N& b) // use two numbers
{
    // ...
    return a + b;
}

int x = 7;
int y = 9;
auto z = plus(x, y);   // z = 18

string xx = "7";
string yy = "9";
auto zz = plus(xx, yy);   // error: string is not a Number
```

Note Concepts with multiple operations have far lower chance of accidentally matching a type than a single-operation concept.

Enforcement

- Flag single-operation `concepts` when used outside the definition of other `concepts`.
- Flag uses of `enable_if` that appears to simulate single-operation `concepts`.

T.21: Define concepts to define complete sets of operations

Reason Improves interoperability. Helps implementers and maintainers.

Example, bad

```
template<typename T> Subtractable = requires(T a, T, b) { a-b; }   // correct syntax?
```

This makes no semantic sense. You need at least `+` to make `-` meaningful and useful.

Examples of complete sets are

- `Arithmetic`: `+`, `-`, `*`, `/`, `+=`, `-=`, `*=`, `/=`
- `Comparable`: `<`, `>`, `<=`, `>=`, `==`, `!=`

Enforcement ???

T.22: Specify axioms for concepts

Reason A meaningful/useful concept has a semantic meaning. Expressing these semantics in an informal, semi-formal, or formal way makes the concept comprehensible to readers and the effort to express it can catch conceptual errors. Specifying semantics is a powerful design tool.

Example

```
template<typename T>
    // The operators +, -, *, and / for a number are assumed to follow the usual mathe
    // axiom(T a, T b) { a + b == b + a; a - a == 0; a * (b + c) == a * b + a * c; /*.
    concept Number = requires(T a, T b) {
        {a + b} -> T;   // the result of a + b is convertible to T
        {a - b} -> T;
        {a * b} -> T;
        {a / b} -> T;
    }
```

Note This is an axiom in the mathematical sense: something that may be assumed without proof. In general, axioms are not provable, and when they are the proof is often beyond the capability of a compiler. An axiom may not be general, but the template writer may assume that it holds for all inputs actually used (similar to a precondition).

Note In this context axioms are Boolean expressions. See the Palo Alto TR for examples. Currently, C++ does not support axioms (even the ISO Concepts TS), so we have to make do with comments for a longish while. Once language support is available, the `//` in front of the axiom can be removed

Note The GSL concepts have well defined semantics; see the Palo Alto TR and the Ranges TS.

Exception Early versions of a new "concept" still under development will often just define simple sets of constraints without a well-specified semantics. Finding good semantics can take effort and time. An incomplete set of constraints can still be very useful:

```
??? binary tree: rotate(), ...
```

A "concept" that is incomplete or without a well-specified semantics can still be useful. However, it should not be assumed to be stable. Each new use case may require such an incomplete concepts to be improved.

Enforcement

- Look for the word "axiom" in concept definition comments

T.23: Differentiate a refined concept from its more general case by adding new use patterns.

Reason Otherwise they cannot be distinguished automatically by the compiler.

Example

```
template<typename I>
concept bool Input_iter = requires (I iter) { ++iter; };

template<typename I>
concept bool Fwd_iter = Input_iter<I> && requires (I iter) { iter++; }
```

The compiler can determine refinement based on the sets of required operations. If two concepts have exactly the same requirements, they are logically equivalent (there is no refinement).

This also decreases the burden on implementers of these types since they do not need any special declarations to "hook into the concept".

Enforcement

- Flag a concept that has exactly the same requirements as another already-seen concept (neither is more refined). To disambiguate them, see T.24.

T.24: Use tag classes or traits to differentiate concepts that differ only in semantics.

Reason Two concepts requiring the same syntax but having different semantics leads to ambiguity unless the programmer differentiates them.

Example

```
template<typename I>    // iterator providing random access
concept bool RA_iter = ...;

template<typename I>    // iterator providing random access to contiguous data
concept bool Contiguous_iter =
    RA_iter<I> && is_contiguous<I>::value;  // ??? why not is_contiguous<I>() or is_con
```

The programmer (in a library) must define `is_contiguous` (a trait) appropriately.

Note Traits can be trait classes or type traits. These can be user-defined or standard-library ones. Prefer the standard-library ones.

Enforcement

- The compiler flags ambiguous use of identical concepts.
- Flag the definition of identical concepts.

T.25: Avoid negating constraints.

Reason Clarity. Maintainability. Functions with complementary requirements expressed using negation are brittle.

Example Initially, people will try to define functions with complementary requirements:

```
template<typename T>
    requires !C<T>    // bad
void f();

template<typename T>
    requires C<T>
void f();
```

This is better:

```
template<typename T>   // general template
    void f();

template<typename T>   // specialization by concept
    requires C<T>
void f();
```

The compiler will choose the unconstrained template only when `C<T>` is unsatisfied. If you do not want to (or cannot) define an unconstrained version of `f()`, then delete it.

```
template<typename T>
void f() = delete;
```

The compiler will select the overload and emit an appropriate error.

Enforcement

- Flag pairs of functions with `C<T>` and `!C<T>` constraints
- Flag all constraint negation

T.26: Prefer to define concepts in terms of use-patterns rather than simple syntax

Reason The definition is more readable and corresponds directly to what a user has to write. Conversions are taken into account. You don't have to remember the names of all the type traits.

Example

```
???
```

Enforcement ???

Template interfaces

???

T.40: Use function objects to pass operations to algorithms

Reason Function objects can carry more information through an interface than a "plain" pointer to function. In general, passing function objects gives better performance than passing pointers to functions.

Example

```
bool greater(double x, double y) { return x>y; }
sort(v, greater);                                  // pointer to function: potentially
sort(v, [](double x, double y) { return x>y; });   // function object
sort(v, greater<>);                                // function object

bool greater_than_7(double x) { return x>7; }
auto x = find_if(v, greater_than_7);               // pointer to function: inflexible
auto y = find_if(v, [](double x) { return x>7; }); // function object: carries the need
auto z = find_if(v, Greater_than<double>(7));      // function object: carries the need
```

You can, of course, gneralize those functions using `auto` or (when and where available) concepts. For example:

```
auto y1 = find_if(v, [](Ordered x) { return x>7; }); // reruire an ordered type
auto z1 = find_if(v, [](auto x) { return x>7; });    // hope that the type has a >
```

Note Lambdas generate function objects.

Note The performance argument depends on compiler and optimizer technology.

Enforcement

- Flag pointer to function template arguments.
- Flag pointers to functions passed as arguments to a template (risk of false positives).

T.41: Require complete sets of operations for a concept

Reason Ease of comprehension. Improved interoperability. Flexibility for template implementers.

Note The issue here is whether to require the minimal set of operations for a template argument (e.g., `==` but not `!=` or `+` but not `+=`). The rule supports the view that a concept should reflect a (mathematically) coherent set of operations.

Example, bad

```
class Minimal {
    // ...
};

bool operator==(const Minimal&,const Minimal&);
bool operator<(const Minimal&,const Minimal&);
Minimal operator+(const Minimal&, const Minimal&);
// no other operators

void f(const Minimal& x, const Minimal& y)
{
    if (!(x==y) { /* ... */ }    // OK
    if (x!=y) { /* ... */ }      //surprise! error

    while (!(x<y)) { /* ... */ }    // OK
    while (x>=y) { /* ... */ }      //surprise! error

    x = x+y;         // OK
    x += y;      // surprise! error
}
```

This is minimal, but surprising and constraining for users. It could even be less efficient.

Example

```
class Convenient {
    // ...
};

bool operator==(const Convenient&,const Convenient&);
bool operator<(const Convenient&,const Convenient&);
// ... and the other comparison operators ...
```

```
Minimal operator+(const Convenient&, const Convenient&);
// .. and the other arithmetic operators ...

void f(const Convenient& x, const Convenient& y)
{
    if (!(x==y) { /* ... */ }     // OK
    if (x!=y) { /* ... */ }       //OK

    while (!(x<y)) { /* ... */ }     // OK
    while (x>=y) { /* ... */ }       //OK

    x = x+y;      // OK
    x += y;       // OK
}
```

It can be a nuisance to define all operators, but not hard. Hopefully, C++17 will give you comparison operators by default.

Enforcement

- Flag classes the support "odd" subsets of a set of operators, e.g., `==` but not `!=` or `+` but not `-`. Yes, `std::string` is "odd", but it's too late to change that.

T.42: Use template aliases to simplify notation and hide implementation details

Reason Improved readability. Implementation hiding. Note that template aliases replace many uses of traits to compute a type. They can also be used to wrap a trait.

Example

```
template<typename T, size_t N>
class matrix {
    // ...
    using Iterator = typename std::vector<T>::iterator;
    // ...
};
```

This saves the user of `Matrix` from having to know that its elements are stored in a `vector` and also saves the user from repeatedly typing `typename std::vector<T>::`.

Example

```
template<typename T>
using Value_type = typename container_traits<T>::value_type;
```

This saves the user of `Value_type` from having to know the technique used to implement `value_types`.

Enforcement

- Flag use of `typename` as a disambiguator outside `using` declarations.
- ???

T.43: Prefer `using` over `typedef` for defining aliases

Reason Improved readability: With `using`, the new name comes first rather than being embedded somewhere in a declaration. Generality: `using` can be used for template aliases, whereas `typedef`s can't easily be templates. Uniformity: `using` is syntactically similar to `auto`.

Example

```
typedef int (*PFI)(int);   // OK, but convoluted

using PFI2 = int (*)(int);   // OK, preferred

template<typename T>
typedef int (*PFT)(T);      // error

template<typename T>
using PFT2 = int (*)(T);   // OK
```

Enforcement

- Flag uses of `typedef`. This will give a lot of "hits" :-(

T.44: Use function templates to deduce class template argument types (where feasible)

Reason Writing the template argument types explicitly can be tedious and unnecessarily verbose.

Example

```
tuple<int, string, double> t1 = {1, "Hamlet", 3.14};   // explicit type
auto t2 = make_tuple(1, "Ophelia"s, 3.14);         // better; deduced type
```

Note the use of the `s` suffix to ensure that the string is a `std::string`, rather than a C-style string.

Note Since you can trivially write a `make_T` function, so could the compiler. Thus, `make_T` functions may become redundant in the future.

Exception Sometimes there isn't a good way of getting the template arguments deduced and sometimes, you want to specify the arguments explicitly:

```
vector<double> v = { 1, 2, 3, 7.9, 15.99 };
list<Record*> lst;
```

Enforcement Flag uses where an explicitly specialized type exactly matches the types of the arguments used.

T.46: Require template arguments to be at least `Regular` or `SemiRegular`

Reason Readability. Preventing surprises and errors. Most uses support that anyway.

Example

```
class X {
        // ...
public:
    explicit X(int);
    X(const X&);            // copy
    X operator=(const X&);
    X(X&&);                 // move
    X& operator=(X&&);
    ~X();
    // ... no moreconstructors ...
};

X x {1};    // fine
X y = x;      // fine
std::vector<X> v(10); // error: no default constructor
```

Note Semiregular requires default constructible.

Enforcement

- Flag types that are not at least `SemiRegular`.

T.47: Avoid highly visible unconstrained templates with common names

Reason An unconstrained template argument is a perfect match for anything so such a template can be preferred over more specific types that require minor conversions. This is particularly annoying/dangerous when ADL is used. Common names make this problem more likely.

Example

```
namespace Bad {
   struct S { int m; };
   template<typename T1, typename T2>
   bool operator==(T1, T2) { cout << "Bad\n"; return true; }
}

namespace T0 {
   bool operator==(int, Bad::S) { cout << "T0\n"; return true; }  // compate to int

   void test()
   {
      Bad::S bad{ 1 };
      vector<int> v(10);
      bool b = 1==bad;
      bool b2 = v.size()==bad;
   }
}
```

This prints `T0` and `Bad`.

Now the `==` in `Bad` was designed to cause trouble, but would you have spotted the problem in real code? The problem is that `v.size()` returns an `unsigned` integer so that a conversion is needed to call the local `==`; the `==` in `Bad` requires no conversions. Realistic types, such as the standard library iterators can be made to exhibit similar anti-social tendencies.

Enforcement ????

T.48: If your compiler does not support concepts, fake them with `enable_if`

Reason ???

Example

```
???
```

Enforcement ???

T.49: Where possible, avoid type-erasure

Reason Type erasure incurs an extra level of indirection by hiding type information behind a separate compilation boundary.

Example

```
???
```

Exceptions: Type erasure is sometimes appropriate, such as for `std::function`.

Enforcement ???

T.50: Avoid writing an unconstrained template in the same namespace as a type

Reason ADL will find the template even when you think it shouldn't.

Example

```
???
```

Note This rule should not be necessary; the committee cannot agree on how to fix ADL, but at least making it not consider unconstrained templates would solve many of the actual problems and remove the need for this rule.

Enforcement ??? unfortunately this will get many false positives; the standard library violates this widely, by putting many unconstrained templates and types into the single namespace `std`

T.def: Template definitions

???

T.60: Minimize a template's context dependencies

Reason Eases understanding. Minimizes errors from unexpected dependencies. Eases tool creation.

Example

```
???
```

Note Having a template operate only on its arguments would be one way of reducing the number of dependencies to a minimum, but that would generally be unmanageable. For example, an algorithm usually uses other algorithms.

Enforcement ??? Tricky

T.61: Do not over-parameterize members (SCARY)

Reason A member that does not depend on a template parameter cannot be used except for a specific template argument. This limits use and typically increases code size.

Example, bad

```
template<typename T, typename A = std::allocator{}>
    // requires Regular<T> && Allocator<A>
class List {
public:
    struct Link {   // does not depend on A
        T elem;
        T* pre;
        T* suc;
    };

    using iterator = Link*;

    iterator first() const { return head; }

    // ...
```

```
private:
    Node* head;
};

List<int> lst1;
List<int, my_allocator> lst2;
```

```
???
```

This looks innocent enough, but ???

```
template<typename T>
struct Link {
    T elem;
    T* pre;
    T* suc;
};

template<typename T, typename A = std::allocator{}>
    // requires Regular<T> && Allocator<A>
class List2 {
public:

    using iterator = Link<T>*;

    iterator first() const { return head; }

    // ...
private:
    Node* head;
};

List<int> lst1;
List<int, my_allocator> lst2;
```

```
???
```

Enforcement

- Flag member types that do not depend on every template argument
- Flag member functions that do not depend on every template argument

T.62: Place non-dependent template members in a non-templated base class

Reason ???

Example

```
template<typename T>
class Foo {
public:
    enum { v1, v2 };
    // ...
};
```

???

```
struct Foo_base {
    enum { v1, v2 };
    // ...
};

template<typename T>
class Foo : public Foo_base {
public:
    // ...
};
```

Note A more general version of this rule would be "If a template class member depends on only N template parameters out of M, place it in a base class with only N parameters." For N == 1, we have a choice of a base class of a class in the surrounding scope as in T.41.

??? What about constants? class statics?

Enforcement

- Flag ???

T.64: Use specialization to provide alternative implementations of class templates

Reason A template defines a general interface. Specialization offers a powerful mechanism for providing alternative implementations of that interface.

Example

```
??? string specialization (==)

??? representation specialization ?
```

Note ???

Enforcement ???

T.65: Use tag dispatch to provide alternative implementations of a function

Reason A template defines a general interface. ???

Example

```
??? that's how we get algorithms like `std::copy` which compiles into a `memmove` call i
```

Note When `concepts` become available such alternatives can be distinguished directly.

Enforcement ???

T.66: Use selection using `enable_if` to optionally define a function

Reason ???

Example

```
???
```

Enforcement ???

T.67: Use specialization to provide alternative implementations for irregular types

Reason ???

Example

```
???
```

Enforcement ???

T.68: Use {} rather than () within templates to avoid ambiguities

Reason ???

Example

```
???
```

Enforcement ???

T.69: Inside a template, don't make an unqualified nonmember function call unless you intend it to be a customization point

Reason To provide only intended flexibility, and avoid accidental environmental changes.

If you intend to call your own helper function `helper(t)` with a value `t` that depends on a template type parameter, put it in a `::detail` namespace and qualify the call as `detail::helper(t);`. Otherwise the call becomes a customization point where any function `helper` in the namespace of `t`'s type can be invoked instead – falling into the second option below, and resulting in problems like unintentionally invoking unconstrained function templates of that name that happen to be in the same namespace as `t`'s type.

There are three major ways to let calling code customize a template.

- Call a member function. Callers can provide any type with such a named member function.

`cpp template<class T> void test(T t) { t.f(); // require T to provide f() }`

- Call a nonmember function without qualification. Callers can provide any type for which there is such a function available in the caller's context or in the namespace of the type.

`cpp template<class T> void test(T t) { f(t); // require f(/*T*/) be available in caller's scope or in T's namespace }`

- Invoke a "trait" – usually a type alias to compute a type, or a `constexpr` function to compute a value, or in rarer cases a traditional traits template to be specialized on the user's type.

`cpp template<class T> void test(T t) { test_traits<T>::f(t); // require customizing test_traits<> to get non-default functions/types test_traits<T>::value_type x; }`

Enforcement

- In a template, flag an unqualified call to a nonmember function that passes a variable of dependent type when there is a nonmember function of the same name in the template's namespace.

T.temp-hier: Template and hierarchy rules:

Templates are the backbone of C++'s support for generic programming and class hierarchies the backbone of its support for object-oriented programming. The two language mechanisms can be used effectively in combination, but a few design pitfalls must be avoided.

T.80: Do not naively templatize a class hierarchy

Reason Templating a class hierarchy that has many functions, especially many virtual functions, can lead to code bloat.

Example, bad

```
template<typename T>
struct Container {         // an interface
    virtual T* get(int i);
    virtual T* first();
    virtual T* next();
    virtual void sort();
};

template<typename T>
class Vector : public Container<T> {
public:
    // ...
};

vector<int> vi;
vector<string> vs;
```

It is probably a dumb idea to define a `sort` as a member function of a container, but it is not unheard of and it makes a good example of what not to do.

Given this, the compiler cannot know if `vector<int>::sort()` is called, so it must generate code for it. Similar for `vector<string>::sort()`. Unless those two functions are called that's code bloat. Imagine what this would do to a class hierarchy with dozens of member functions and dozens of derived classes with many instantiations.

Note In many cases you can provide a stable interface by not parameterizing a base; see Rule.

Enforcement

- Flag virtual functions that depend on a template argument. ??? False positives

T.81: Do not mix hierarchies and arrays

Reason An array of derived classes can implicitly "decay" to a pointer to a base class with potential disastrous results.

Example Assume that `Apple` and `Pear` are two kinds of `Fruit`s.

```
void maul(Fruit* p)
{
    *p = Pear{};     // put a Pear into *p
    p[1] = Pear{};   // put a Pear into p[2]
}

Apple aa [] = { an_apple, another_apple };   // aa contains Apples (obviously!)

maul(aa);
Apple& a0 = &aa[0];   // a Pear?
Apple& a1 = &aa[1];   // a Pear?
```

Probably, `aa[0]` will be a `Pear` (without the use of a cast!). If `sizeof(Apple) != sizeof(Pear)` the access to `aa[1]` will not be aligned to the proper start of an object in the array. We have a type violation and possibly (probably) a memory corruption. Never write such code.

Note that `maul()` violates the a `T*` points to an individual object Rule.

Alternative: Use a proper container:

```
void maul2(Fruit* p)
{
    *p = Pear{};   // put a Pear into *p
}

vector<Apple> va = { an_apple, another_apple };   // aa contains Apples (obviously!)

maul2(aa);       // error: cannot convert a vector<Apple> to a Fruit*
maul2(&aa[0]);   // you asked for it
```

```
Apple& a0 = &aa[0];    // a Pear?
```

Note that the assignment in `maul2()` violated the no-slicing Rule.

Enforcement

- Detect this horror!

T.82: Linearize a hierarchy when virtual functions are undesirable

Reason ???

Example

```
???
```

Enforcement ???

T.83: Do not declare a member function template virtual

Reason C++ does not support that. If it did, vtbls could not be generated until link time. And in general, implementations must deal with dynamic linking.

Example, don't

```
class Shape {
    // ...
    template<class T>
    virtual bool intersect(T* p);    // error: template cannot be virtual
};
```

Note We need a rule because people keep asking about this

Alternative Double dispatch, visitors, calculate which function to call

Enforcement The compiler handles that.

T.84: Use a non-template core implementation to provide an ABI-stable interface

Reason Improve stability of code. Avoids code bloat.

Example It could be a base class:

```
struct Link_base {   // stable
    Link* suc;
    Link* pre;
};

template<typename T>   // templated wrapper to add type safety
struct Link : Link_base {
    T val;
};

struct List_base {
    Link_base* first;   // first element (if any)
    int sz;             // number of elements
    void add_front(Link_base* p);
    // ...
};

template<typename T>
class List : List_base {
public:
    void put_front(const T& e) { add_front(new Link<T>{e}); }   // implicit cast to Lin
    T& front() { static_cast<Link<T>*>(first).val; }   // explicit cast back to Link<T>
    // ...
};

List<int> li;
List<string> ls;
```

Now there is only one copy of the operations linking and unlinking elements of a `List`. The `Link` and `List` classes does nothing but type manipulation.

Instead of using a separate "base" type, another common technique is to specialize for `void` or `void*` and have the general template for `T` be just the safely-encapsulated casts to and from the core `void` implementation.

Alternative: Use a PIMPL implementation.

Enforcement ???

T.var: Variadic template rules

???

T.100: Use variadic templates when you need a function that takes a variable number of arguments of a variety of types

Reason Variadic templates is the most general mechanism for that, and is both efficient and type-safe. Don't use C varargs.

Example

```
??? printf
```

Enforcement

```
* Flag uses of `va_arg` in user code.
```

T.101: ??? How to pass arguments to a variadic template ???

Reason ???

Example

```
??? beware of move-only and reference arguments
```

Enforcement ???

T.102: How to process arguments to a variadic template

Reason ???

Example

```
??? forwarding, type checking, references
```

Enforcement ???

T.103: Don't use variadic templates for homogeneous argument lists

Reason There are more precise ways of specifying a homogeneous sequence, such as an `initializer_list`.

Example

```
???
```

Enforcement ???

T.meta: Template metaprogramming (TMP)

Templates provide a general mechanism for compile-time programming.

Metaprogramming is programming where at least one input or one result is a type. Templates offer Turing-complete (modulo memory capacity) duck typing at compile time. The syntax and techniques needed are pretty horrendous.

T.120: Use template metaprogramming only when you really need to

Reason Template metaprogramming is hard to get right, slows down compilation, and is often very hard to maintain. However, there are real-world examples where template metaprogramming provides better performance that any alternative short of expert-level assembly code. Also, there are real-world examples where template metaprogramming expresses the fundamental ideas better than run-time code. For example, if you really need AST manipulation at compile time (e.g., for optional matrix operation folding) there may be no other way in C++.

Example, bad

```
???
```

Example, bad

```
enable_if
```

Instead, use concepts. But see How to emulate concepts if you don't have language support.

Example

```
??? good
```

Alternative: If the result is a value, rather than a type, use a `constexpr` function.

Note If you feel the need to hide your template metaprogramming in macros, you have probably gone too far.

T.121: Use template metaprogramming primarily to emulate concepts

Reason Until concepts become generally available, we need to emulate them using TMP. Use cases that require concepts (e.g. overloading based on concepts) are among the most common (and simple) uses of TMP.

Example

```
template<typename Iter>
    /*requires*/ enable_if<random_access_iterator<Iter>, void>
advance(Iter p, int n) { p += n; }

template<typename Iter>
    /*requires*/ enable_if<forward_iterator<Iter>, void>
advance(Iter p, int n) { assert(n >= 0); while (n--) ++p;}
```

Note Such code is much simpler using concepts:

```
void advance(RandomAccessIterator p, int n) { p += n; }

void advance(ForwardIterator p, int n) { assert(n >= 0); while (n--) ++p;}
```

Enforcement ???

T.122: Use templates (usually template aliases) to compute types at compile time

Reason Template metaprogramming is the only directly supported and half-way principled way of generating types at compile time.

Note "Traits" techniques are mostly replaced by template aliases to compute types and `constexpr` functions to compute values.

Example

```
??? big object / small object optimization
```

Enforcement ???

T.123: Use `constexpr` functions to compute values at compile time

Reason A function is the most obvious and conventional way of expressing the computation of a value. Often a `constexpr` function implies less compile-time overhead than alternatives.

Note "Traits" techniques are mostly replaced by template aliases to compute types and `constexpr` functions to compute values.

Example

```
template<typename T>
    // requires Number<T>
constexpr T pow(T v, int n)   // power/exponential
{
    T res = 1;
    while (n--) res *= v;
    return res;
}

constexpr auto f7 = pow(pi, 7);
```

Enforcement

```
* Flag template metaprograms yielding a value. These should be replaced with `constexpr`
```

T.124: Prefer to use standard-library TMP facilities

Reason Facilities defined in the standard, such as `conditional`, `enable_if`, and `tuple`, are portable and can be assumed to be known.

Example

```
???
```

Enforcement ???

T.125: If you need to go beyond the standard-library TMP facilities, use an existing library

Reason Getting advanced TMP facilities is not easy and using a library makes you part of a (hopefully supportive) community. Write your own "advanced TMP support" only if you really have to.

Example

```
???
```

Enforcement ???

Other template rules

T.140: Name all nontrivial operations

Reason Documentation, readability, opportunity for reuse.

Example

```
???
```

Example, good

```
???
```

Note whether functions, lambdas, or operators.

Exceptions

- Lambdas logically used only locally, such as an argument to `for_each` and similar control flow algorithms.
- Lambdas as initializers

Enforcement ???

T.141: Use an unnamed lambda if you need a simple function object in one place only

Reason That makes the code concise and gives better locality than alternatives.

Example

```
auto earlyUsersEnd = std::remove_if(users.begin(), users.end(),
                                    [](const User &a) { return a.id > 100; });
```

Exception: Naming a lambda can be useful for clarity even if it is used only once

Enforcement

- Look for identical and near identical lambdas (to be replaced with named functions or named lambdas).

T.142?: Use template variables to simplify notation

Reason Improved readability.

Example

```
???
```

Enforcement ???

T.143: Don't write unintentionally nongeneric code

Reason Generality. Reusability. Don't gratuitously commit to details; use the most general facilities available.

Example Use `!=` instead of `<` to compare iterators; `!=` works for more objects because it doesn't rely on ordering.

```
for (auto i = first; i < last; ++i) {   // less generic
    // ...
}

for (auto i = first; i != last; ++i) {   // good; more generic
    // ...
}
```

Of course, range-for is better still where it does what you want.

Example Use the least-derived class that has the functionality you need.

```
class base {
public:
    void f();
    void g();
};

class derived1 : public base {
public:
    void h();
};

class derived2 : public base {
public:
    void j();
};

void myfunc(derived1& param)  // bad, unless there is a specific reason for limiting to
{
    use(param.f());
    use(param.g());
}

void myfunc(base& param)   // good, uses only base interface so only commit to that
{
    use(param.f());
    use(param.g());
}
```

Enforcement

- Flag comparison of iterators using `<` instead of `!=`.
- Flag `x.size() == 0` when `x.empty()` or `x.is_empty()` is available. Emptiness works for more containers than size(), because some containers don't know their size or are conceptually of unbounded size.
- Flag functions that take a pointer or reference to a more-derived type but only use functions declared in a base type.

T.144: Don't specialize function templates

Reason You can't partially specialize a function template per language rules. You can fully specialize a function template but you almost certainly want to overload instead –

because function template specializations don't participate in overloading, they don't act as you probably wanted. Rarely, you should actually specialize by delegating to a class template that you can specialize properly.

Example

```
???
```

Exceptions: If you do have a valid reason to specialize a function template, just write a single function template that delegates to a class template, then specialize the class template (including the ability to write partial specializations).

Enforcement

- Flag all specializations of a function template. Overload instead.

CPL: C-style programming

C and C++ are closely related languages. They both originate in "Classic C" from 1978 and have evolved in ISO committees since then. Many attempts have been made to keep them compatible, but neither is a subset of the other.

C rule summary:

- CPL.1: Prefer C++ to C
- CPL.2: If you must use C, use the common subset of C and C++, and compile the C code as C++
- CPL.3: If you must use C for interfaces, use C++ in the code using such interfaces

CPL.1: Prefer C++ to C

Reason C++ provides better type checking and more notational support. It provides better support for high-level programming and often generates faster code.

Example

```
char ch = 7;
void* pv = &ch;
int* pi = pv;    // not C++
*pi = 999;       // overwrite sizeof(int) bytes near &ch
```

Enforcement Use a C++ compiler.

CPL.2: If you must use C, use the common subset of C and C++, and compile the C code as C++

Reason That subset can be compiled with both C and C++ compilers, and when compiled as C++ is better type checked than "pure C."

Example

```
int* p1 = malloc(10 * sizeof(int));                      // not C++
int* p2 = static_cast<int*>(malloc(10 * sizeof(int)));   // not C, C-style C++
int* p3 = new int[10];                                   // not C
int* p4 = (int*)malloc(10 * sizeof(int));                // both C and C++
```

Enforcement

- Flag if using a build mode that compiles code as C.
- The C++ compiler will enforce that the code is valid C++ unless you use C extension options.

CPL.3: If you must use C for interfaces, use C++ in the calling code using such interfaces

Reason C++ is more expressive than C and offers better support for many types of programming.

Example For example, to use a 3rd party C library or C systems interface, define the low-level interface in the common subset of C and C++ for better type checking. Whenever possible encapsulate the low-level interface in an interface that follows the C++ guidelines (for better abstraction, memory safety, and resource safety) and use that C++ interface in C++ code.

Example You can call C from C++:

```
// in C:
double sqrt(double);

// in C++:
extern "C" double sqrt(double);

sqrt(2);
```

Example You can call C++ from C:

```
// in C:
X call_f(struct Y*, int);

// in C++:
extern "C" X call_f(Y* p, int i)
{
    return p->f(i);    // possibly a virtual function call
}
```

Enforcement None needed

SF: Source files

Distinguish between declarations (used as interfaces) and definitions (used as implementations). Use header files to represent interfaces and to emphasize logical structure.

Source file rule summary:

- SF.1: Use a `.cpp` suffix for code files and `.h` for interface files if your project doesn't already follow another convention
- SF.2: A `.h` file may not contain object definitions or non-inline function definitions
- SF.3: Use `.h` files for all declarations used in multiple source files
- SF.4: Include `.h` files before other declarations in a file
- SF.5: A `.cpp` file must include the `.h` file(s) that defines its interface
- SF.6: Use `using`-directives for transition, for foundation libraries (such as `std`), or within a local scope
- SF.7: Don't put a `using`-directive in a header file
- SF.8: Use `#include` guards for all `.h` files
- SF.9: Avoid cyclic dependencies among source files
- SF.20: Use `namespace`s to express logical structure
- SF.21: Don't use an unnamed (anonymous) namespace in a header
- SF.22: Use an unnamed (anonymous) namespace for all internal/nonexported entities

SF.1: Use a `.cpp` suffix for code files and `.h` for interface files if your project doesn't already follow another convention

Reason It's a longstanding convention. But consistency is more important, so if your project uses something else, follow that.

Note This convention reflects a common use pattern: Headers are more often shared with C to compile as both C++ and C, which typically uses `.h`, and it's easier to name all headers `.h` instead of having different extensions for just those headers that are intended to be shared with C. On the other hand, implementation files are rarely shared with C and so should typically be distinguished from `.c` files, so it's normally best to name all C++ implementation files something else (such as `.cpp`).

The specific names `.h` and `.cpp` are not required (just recommended as a default) and other names are in widespread use. Examples are `.hh` and `.cxx`. Use such names equivalently. In this document we refer to `.h` and `.cpp` as a shorthand for header and implementation files, even though the actual extension may be different.

Example

```
// foo.h:
extern int a;   // a declaration
extern void foo();

// foo.cpp:
int a;   // a definition
void foo() { ++a; }
```

`foo.h` provides the interface to `foo.cpp`. Global variables are best avoided.

Example, bad

```
// foo.h:
int a;   // a definition
void foo() { ++a; }
```

`#include<foo.h>` twice in a program and you get a linker error for two one-definition-rule violations.

Enforcement

- Flag non-conventional file names.
- Check that `.h` and `.cpp` (and equivalents) follow the rules below.

SF.2: A `.h` file may not contain object definitions or non-inline function definitions

Reason Including entities subject to the one-definition rule leads to linkage errors.

Example

```
???
```

Alternative formulation: A `.h` file must contain only:

- `#include`s of other `.h` files (possibly with include guards)
- templates
- class definitions
- function declarations
- `extern` declarations
- `inline` function definitions
- `constexpr` definitions
- `const` definitions
- `using` alias definitions
- ???

Enforcement Check the positive list above.

SF.3: Use `.h` files for all declarations used in multiple source files

Reason Maintainability. Readability.

Example, bad

```
// bar.cpp:
void bar() { cout << "bar\n"; }

// foo.cpp:
extern void bar();
void foo() { bar(); }
```

A maintainer of `bar` cannot find all declarations of `bar` if its type needs changing. The user of `bar` cannot know if the interface used is complete and correct. At best, error messages come (late) from the linker.

Enforcement

- Flag declarations of entities in other source files not placed in a `.h`.

SF.4: Include `.h` files before other declarations in a file

Reason Minimize context dependencies and increase readability.

Example

```
#include<vector>
#include<algorithm>
#include<string>

// ... my code here ...
```

Example, bad

```
#include<vector>

// ... my code here ...

#include<algorithm>
#include<string>
```

Note This applies to both `.h` and `.cpp` files.

Exception: Are there any in good code?

Enforcement Easy.

SF.5: A `.cpp` file must include the `.h` file(s) that defines its interface

Reason This enables the compiler to do an early consistency check.

Example, bad

```
// foo.h:
void foo(int);
int bar(long double);
int foobar(int);
```

```
// foo.cpp:
void foo(int) { /* ... */ }
int bar(double) { /* ... */ }
double foobar(int);
```

The errors will not be caught until link time for a program calling `bar` or `foobar`.

Example

```
// foo.h:
void foo(int);
int bar(long double);
int foobar(int);

// foo.cpp:
#include<foo.h>

void foo(int) { /* ... */ }
int bar(double) { /* ... */ }
double foobar(int);   // error: wrong return type
```

The return-type error for `foobar` is now caught immediately when `foo.cpp` is compiled. The argument-type error for `bar` cannot be caught until link time because of the possibility of overloading, but systematic use of `.h` files increases the likelihood that it is caught earlier by the programmer.

Enforcement ???

SF.6: Use `using`-directives for transition, for foundation libraries (such as `std`), or within a local scope

Reason ???

Example

```
???
```

Enforcement ???

SF.7: Don't put a `using`-directive in a header file

Reason Doing so takes away an `#include`r's ability to effectively disambiguate and to use alternatives.

Example

```
???
```

Enforcement ???

SF.8: Use `#include` guards for all `.h` files

Reason To avoid files being `#include`d several times.

Example

```
// file foobar.h:
#ifndef FOOBAR_H
#define FOOBAR_H
// ... declarations ...
#endif // FOOBAR_H
```

Enforcement Flag `.h` files without `#include` guards.

SF.9: Avoid cyclic dependencies among source files

Reason Cycles complicates comprehension and slows down compilation. Complicates conversion to use language-supported modules (when they become available).

Note Eliminate cycles; don't just break them with `#include` guards.

Example, bad

```
// file1.h:
#include "file2.h"

// file2.h:
#include "file3.h"

// file3.h:
#include "file1.h"
```

Enforcement Flag all cycles.

SF.20: Use `namespace`s to express logical structure

Reason ???

Example

```
???
```

Enforcement ???

SF.21: Don't use an unnamed (anonymous) namespace in a header

Reason It is almost always a bug to mention an unnamed namespace in a header file.

Example

```
???
```

Enforcement

- Flag any use of an anonymous namespace in a header file.

SF.22: Use an unnamed (anonymous) namespace for all internal/nonexported entities

Reason Nothing external can depend on an entity in a nested unnamed namespace. Consider putting every definition in an implementation source file in an unnamed namespace unless that is defining an "external/exported" entity.

Example An API class and its members can't live in an unnamed namespace; but any "helper" class or function that is defined in an implementation source file should be at an unnamed namespace scope.

```
???
```

Enforcement

- ???

SL: The Standard Library

Using only the bare language, every task is tedious (in any language). Using a suitable library any task can be reasonably simple.

Standard-library rule summary:

- SL.1: Use libraries wherever possible
- SL.2: Prefer the standard library to other libraries
- ???

SL.1: Use libraries wherever possible

Reason Save time. Don't re-invent the wheel. Don't replicate the work of others. Benefit from other people's work when they make improvements. Help other people when you make improvements.

SL.2: Prefer the standard library to other libraries

Reason More people know the standard library. It is more likely to be stable, well-maintained, and widely available than your own code or most other libraries.

SL.con: Containers

- SL.10: Prefer using STL `array` or `vector` instead of a C array
- SL.11: Prefer using STL `vector` by default unless you have a reason to use a different container ???

SL.10: Prefer using STL `array` or `vector` instead of a C array

Reason C arrays are less safe, and have no advantages over `array` and `vector`. For a fixed-length array, use `std::array`, which does not degenerate to a pointer when passed to a function and does know its size. For a variable-length array, use `std::vector`, which additionally can change its size and handles memory allocation.

Example

```
int v[SIZE];                        // BAD

std::array<int,SIZE> w;             // ok
```

Example

```
int* v = new int[initial_size];      // BAD, owning raw pointer
delete[] v;                          // BAD, manual delete

std::vector<int> w(initial_size);    // ok
```

Enforcement

- Flag declaration of a C array inside a function or class that also declares an STL container (to avoid excessive noisy warnings on legacy non-STL code). To fix: At least change the C array to a `std::array`.

SL.11: Prefer using STL `vector` by default unless you have a reason to use a different container

Reason `vector` and `array` are the only standard containers that offer the fastest general-purpose access (random access, including being vectorization-friendly), the fastest default access pattern (begin-to-end or end-to-begin is prefetcher-friendly), and the lowest space overhead (contiguous layout has zero per-element overhead, which is cache-friendly). Usually you need to add and remove elements from the container, so use `vector` by default; if you don't need to modify the container's size, use `array`.

Even when other containers seem more suited, such a `map` for O(log N) lookup performance or a `list` for efficient insertion in the middle, a `vector` will usually still perform better for containers up to a few KB in size.

Note `string` should not be used as a container of individual characters. A `string` is a textual string; if you want a container of characters, use `vector</*char_type*/>` or `array</*char_type*/>` instead.

Exceptions If you have a good reason to use another container, use that instead. For example:

- If `vector` suits your needs but you don't need the container to be variable size, use `array` instead.
- If you want a dictionary-style lookup container that guarantees O(K) or O(log N) lookups, the container will be larger (more than a few KB) and you perform frequent inserts so that the overhead of maintaining a sorted `vector` is infeasible, go ahead and use an `unordered_map` or `map` instead.

Enforcement

- Flag a `vector` whose size never changes after construction (such as because it's `const` or because no non-`const` functions are called on it). To fix: Use an `array` instead.

SL.str: String

???

SL.io: Iostream

???

SL.???: Use character-level input only when you have to; *expr.low*.

SL.???: When reading, always consider ill-formed input; *expr.low*.

SL.50: Avoid `endl`

Reason

The `endl` manipulator is mostly equivalent to `'\\n'` and `"\\n"`; as most commonly used it simply slows down output by doing redundant `flush()`s. This slowdown can be significant compared to `printf`-style output.

Example

```
cout << "Hello, World!" << endl;     // two output operations and a flush
cout << "hello, World!\n";           // one output operation and no flush
```

Note For `cin`/`cout` (and equivalent) interaction, there is no reason to flush; that's done automatically. For writing to a file, there is rarely a need to `flush`.

Note Apart from the (occasionally important) issue of performance, the choice between `"\\n"` and `endl` is almost completely aesthetic.

SL.regex: Regex

???

SL:c: The C standard library

SL.???: C-style strings

SL.???: printf/scanf

A: Architectural Ideas

This section contains ideas about ???

A.1 Separate stable from less stable part of code

???

A.2 Express potentially reusable parts as a library

???

A.3 Express potentially separately maintained parts as a library

???

Non-Rules and myths

This section contains rules and guidelines that are popular somewhere, but that we deliberately don't recommend. In the context of the styles of programming we recommend and support with the guidelines, these "non-rules" would do harm.

Non-rule summary:

- all declarations on top of function
- single-return rule
- no exceptions
- one class per source file
- two-phase initialization
- goto exit

RF: References

Many coding standards, rules, and guidelines have been written for C++, and especially for specialized uses of C++. Many

- focus on lower-level issues, such as the spelling of identifiers
- are written by C++ novices
- see "stopping programmers from doing unusual things" as their primary aim
- aim at portability across many compilers (some 10 years old)
- are written to preserve decades old code bases
- aim at a single application domain
- are downright counterproductive
- are ignored (must be ignored by programmers to get their work done well)

A bad coding standard is worse than no coding standard. However an appropriate set of guidelines are much better than no standards: "Form is liberating."

Why can't we just have a language that allows all we want and disallows all we don't want ("a perfect language")? Fundamentally, because affordable languages (and their tool chains) also serve people with needs that differ from yours and serve more needs than you have today. Also, your needs change over time and a general-purpose language is needed to allow you to adapt. A language that is ideal for today would be overly restrictive tomorrow.

Coding guidelines adapt the use of a language to specific needs. Thus, there cannot be a single coding style for everybody. We expect different organizations to provide additions, typically with more restrictions and firmer style rules.

Reference sections:

- RF.rules: Coding rules
- RF.books: Books with coding guidelines
- RF.C++: C++ Programming (C++11/C++14)
- RF.web: Websites
- RS.video: Videos about "modern C++"
- RF.man: Manuals

RF.rules: Coding rules

- Boost Library Requirements and Guidelines. ???.
- Bloomberg: BDE C++ Coding. Has a strong emphasis on code organization and layout.
- Facebook: ???
- GCC Coding Conventions. C++03 and (reasonably) a bit backwards looking.

- Google C++ Style Guide. Too timid and reflects its 1990s origins. A critique from 2014. Google are busy updating their code base and we don't know how accurately the posted guideline reflects their actual code. This set of recommendations is evolving.
- JSF++: JOINT STRIKE FIGHTER AIR VEHICLE C++ CODING STANDARDS. Document Number 2RDU00001 Rev C. December 2005. For flight control software. For hard real time. This means that it is necessarily very restrictive ("if the program fails somebody dies"). For example, no free store allocation or deallocation may occur after the plane takes off (no memory overflow and no fragmentation allowed). No exception may be used (because there was no available tool for guaranteeing that an exception would be handled within a fixed short time). Libraries used have to have been approved for mission critical applications. Any similarities to this set of guidelines are unsurprising because Bjarne Stroustrup was an author of JSF++. Recommended, but note its very specific focus.
- Mozilla Portability Guide. As the name indicates, this aims for portability across many (old) compilers. As such, it is restrictive.
- Geosoft.no: C++ Programming Style Guidelines. ???.
- Possibility.com: C++ Coding Standard. ???.
- SEI CERT: Secure C++ Coding Standard. A very nicely done set of rules (with examples and rationales) done for security-sensitive code. Many of their rules apply generally.
- High Integrity C++ Coding Standard.
- llvm. Somewhat brief, pre-C++11, and (not unreasonably) adjusted to its domain.
- ???

RF.books: Books with coding guidelines

- Meyers96 Scott Meyers: *More Effective C++*. Addison-Wesley 1996.
- Meyers97 Scott Meyers: *Effective C++, Second Edition*. Addison-Wesley 1997.
- Meyers01 Scott Meyers: *Effective STL*. Addison-Wesley 2001.
- Meyers05 Scott Meyers: *Effective C++, Third Edition*. Addison-Wesley 2005.
- Meyers15 Scott Meyers: *Effective Modern C++*. O'Reilly 2015.
- SuttAlex05 Sutter and Alexandrescu: *C++ Coding Standards*. Addison-Wesley 2005. More a set of meta-rules than a set of rules. Pre-C++11.
- Stroustrup05 Bjarne Stroustrup: A rationale for semantically enhanced library languages. LCSD05. October 2005.
- Stroustrup14 Stroustrup: A Tour of C++. Addison Wesley 2014. Each chapter ends with an advice section consisting of a set of recommendations.
- Stroustrup13 Stroustrup: The C++ Programming Language (4th Edition). Addison Wesley 2013. Each chapter ends with an advice section consisting of a set of recommendations.
- Stroustrup: Style Guide for Programming: Principles and Practice using C++. Mostly low-level naming and layout rules. Primarily a teaching tool.

RF.C++: C++ Programming (C++11/C++14)

- TC++PL4
- Tour++
- Programming: Principles and Practice using C++

RF.web: Websites

- isocpp.org
- Bjarne Stroustrup's home pages
- WG21
- Boost
- Adobe open source
- Poco libraries

RS.video: Videos about "modern C++"

- Bjarne Stroustrup: C++11 Style. 2012.
- Bjarne Stroustrup: The Essence of C++: With Examples in C++84, C++98, C++11, and C++14. 2013
- All the talks from CppCon '14
- Bjarne Stroustrup: The essence of C++ at the University of Edinburgh. 2014.
- Sutter: ???
- ??? more ???

RF.man: Manuals

- ISO C++ Standard C++11
- ISO C++ Standard C++14
- Palo Alto "Concepts" TR
- ISO C++ Concepts TS
- WG21 Ranges report

Acknowledgements

Thanks to the many people who contributed rules, suggestions, supporting information, references, etc.:

- Peter Juhl
- Neil MacIntosh

- Axel Naumann
- Andrew Pardoe
- Gabriel Dos Reis
- Zhuang, Jiangang (Jeff)
- Sergey Zubkov

Profiles

A "profile" is a set of deterministic and portably enforceable subset rules (i.e., restrictions) that are designed to achieve a specific guarantee. "Deterministic" means they require only local analysis and could be implemented in a compiler (though they don't need to be). "Portably enforceable" means they are like language rules, so programmers can count on enforcement tools giving the same answer for the same code.

Code written to be warning-free using such a language profile is considered to conform to the profile. Conforming code is considered to be safe by construction with regard to the safety properties targeted by that profile. Conforming code will not be the root cause of errors for that property, although such errors may be introduced into a program by other code, libraries or the external environment. A profile may also introduce additional library types to ease conformance and encourage correct code.

Profiles summary:

- Pro.type: Type safety
- Pro.bounds: Bounds safety
- Pro.lifetime: Lifetime safety

Type safety profile

This profile makes it easier to construct code that uses types correctly and avoids inadvertent type punning. It does so by focusing on removing the primary sources of type violations, including unsafe uses of casts and unions.

For the purposes of this section, type-safety is defined to be the property that a program does not use a variable as a type it is not. Memory accessed as a type `T` should not be valid memory that actually contains an object of an unrelated type `U`. (Note that the safety is intended to be complete when combined also with Bounds safety and Lifetime safety.)

The following are under consideration but not yet in the rules below, and may be better in other profiles:

- narrowing arithmetic promotions/conversions (likely part of a separate safe-arithmetic profile)
- arithmetic cast from negative floating point to unsigned integral type (ditto)

- selected undefined behavior: ??? this is a big bucket, start with Gaby's UB list
- selected unspecified behavior: ??? would this really be about safety, or more a portability concern?
- constness violations? if we rely on it for safety

An implementation of this profile shall recognize the following patterns in source code as non-conforming and issue a diagnostic.

Type.1: Don't use `reinterpret_cast`.

Reason Use of these casts can violate type safety and cause the program to access a variable that is actually of type `X` to be accessed as if it were of an unrelated type `Z`.

Example, bad

```
std::string s = "hello world";
double* p = reinterpret_cast<double*>(&s); // BAD
```

Enforcement Issue a diagnostic for any use of `reinterpret_cast`. To fix: Consider using a `variant` instead.

Type.2: Don't use `static_cast` downcasts. Use `dynamic_cast` instead.

Reason Use of these casts can violate type safety and cause the program to access a variable that is actually of type `X` to be accessed as if it were of an unrelated type `Z`.

Example, bad

```
class base { public: virtual ~base() = 0; };

class derived1 : public base { };

class derived2 : public base {
    std::string s;
public:
    std::string get_s() { return s; }
};

derived1 d1;
base* p = &d1; // ok, implicit conversion to pointer to base is fine

derived2* p2 = static_cast<derived2*>(p); // BAD, tries to treat d1 as a derived2, whic
cout << p2->get_s(); // tries to access d1's nonexistent string member, instead sees a
```

Example, bad

```
struct Foo { int a, b; };
struct Foobar : Foo { int bar; };

void use(int i, Foo& x)
{
    if (0<i) {
        Foobar& x1 = dynamic_cast<Foobar&>(x);  // error: Foo is not polymorphic
        Foobar& x2 = static_cast<Foobar&>(x);   // bad
        // ...
    }
    // ...
}

// ...

use(99, *new Foo{1, 2});  // not a Foobar
```

If a class hierarchy isn't polymorphic, avoid casting. It is entirely unsafe. Look for a better design. See also C.146.

Enforcement Issue a diagnostic for any use of `static_cast` to downcast, meaning to cast from a pointer or reference to `X` to a pointer or reference to a type that is not `X` or an accessible base of `X`. To fix: If this is a downcast or cross-cast then use a `dynamic_cast` instead, otherwise consider using a `variant` instead.

Type.3: Don't use `const_cast` to cast away `const` (i.e., at all).

Reason Casting away `const` is a lie. If the variable is actually declared `const`, it's a lie punishable by undefined behavior.

Example, bad

```
void f(const int& i)
{
    const_cast<int&>(i) = 42;   // BAD
}

static int i = 0;
static const int j = 0;

f(i); // silent side effect
f(j); // undefined behavior
```

Example Sometimes you may be tempted to resort to `const_cast` to avoid code duplication, such as when two accessor functions that differ only in `const`-ness have similar implementations. For example:

```
class bar;

class foo {
    bar mybar;
public:                         // BAD, duplicates logic
          bar& get_bar()       { /* complex logic around getting a non-const reference
    const bar& get_bar() const { /* same complex logic around getting a const reference
};
```

Instead, prefer to share implementations. Normally, you can just have the non-`const` function call the `const` function. However, when there is complex logic this can lead to the following pattern that still resorts to a `const_cast`:

```
class foo {
    bar mybar;
public:                         // not great, non-const calls const version but resorts
          bar& get_bar()       { return const_cast<bar&>(static_cast<const foo&>(*this)
    const bar& get_bar() const { /* the complex logic around getting a const reference
};
```

Although this pattern is safe when applied correctly, because the caller must have had a non-`const` object to begin with, it's not ideal because the safety is hard to enforce automatically as a checker rule.

Instead, prefer to put the common code in a common helper function – and make it a template so that it deduces `const`. This doesn't use any `const_cast` at all:

```
class foo {
    bar mybar;

    template<class T>           // good, deduces whether T is const or non-const
    static auto get_bar_impl(T& t) -> decltype(t.get_bar())
        { /* the complex logic around getting a possibly-const reference to mybar */ }

public:                         // good
          bar& get_bar()       { return get_bar_impl(*this); }
    const bar& get_bar() const { return get_bar_impl(*this); }
};
```

Exception: You may need to cast away `const` when calling `const`-incorrect functions. Prefer to wrap such functions in inline `const`-correct wrappers to encapsulate the cast in one place.

Enforcement Issue a diagnostic for any use of `const_cast`. To fix: Either don't use the variable in a non-`const` way, or don't make it `const`.

Type.4: Don't use C-style `(T)expression` casts that would perform a `static_cast` downcast, `const_cast`, or `reinterpret_cast`.

Reason Use of these casts can violate type safety and cause the program to access a variable that is actually of type `X` to be accessed as if it were of an unrelated type `Z`. Note that a C-style `(T)expression` cast means to perform the first of the following that is possible: a `const_cast`, a `static_cast`, a `static_cast` followed by a `const_cast`, a `reinterpret_cast`, or a `reinterpret_cast` followed by a `const_cast`. This rule bans `(T)expression` only when used to perform an unsafe cast.

Example, bad

```
std::string s = "hello world";
double* p = (double*)(&s); // BAD

class base { public: virtual ~base() = 0; };

class derived1 : public base { };

class derived2 : public base {
    std::string s;
public:
    std::string get_s() { return s; }
};

derived1 d1;
base* p = &d1; // ok, implicit conversion to pointer to base is fine

derived2* p2 = (derived2*)(p); // BAD, tries to treat d1 as a derived2, which it is not
cout << p2->get_s(); // tries to access d1's nonexistent string member, instead sees ar

void f(const int& i) {
    (int&)(i) = 42;   // BAD
}

static int i = 0;
static const int j = 0;

f(i); // silent side effect
f(j); // undefined behavior
```

Enforcement Issue a diagnostic for any use of a C-style `(T)expression` cast that would invoke a `static_cast` downcast, `const_cast`, or `reinterpret_cast`. To fix: Use a `dynamic_cast`, `const`-correct declaration, or `variant`, respectively.

Type.5: Don't use a variable before it has been initialized.

ES.20: Always initialize an object is required.

Type.6: Always initialize a member variable.

Reason Before a variable has been initialized, it does not contain a deterministic valid value of its type. It could contain any arbitrary bit pattern, which could be different on each call.

Example

```
struct X { int i; };

X x;
use(x); // BAD, x has not been initialized

X x2{}; // GOOD
use(x2);
```

Enforcement

- Issue a diagnostic for any constructor of a non-trivially-constructible type that does not initialize all member variables. To fix: Write a data member initializer, or mention it in the member initializer list.
- Issue a diagnostic when constructing an object of a trivially constructible type without `()` or `{}` to initialize its members. To fix: Add `()` or `{}`.

Type.7: Avoid accessing members of raw unions. Prefer `variant` instead.

Reason Reading from a union member assumes that member was the last one written, and writing to a union member assumes another member with a nontrivial destructor had its destructor called. This is fragile because it cannot generally be enforced to be safe in the language and so relies on programmer discipline to get it right.

Example

```
union U { int i; double d; };

U u;
u.i = 42;
use(u.d); // BAD, undefined

variant<int, double> u;
u = 42; // u now contains int
use(u.get<int>()); // ok
use(u.get<double>()); // throws ??? update this when standardization finalizes the var
```

Note that just copying a union is not type-unsafe, so safe code can pass a union from one piece of unsafe code to another.

Enforcement

- Issue a diagnostic for accessing a member of a union. To fix: Use a `variant` instead.

Type.8: Avoid reading from varargs or passing vararg arguments. Prefer variadic template parameters instead.

Reason Reading from a vararg assumes that the correct type was actually passed. Passing to varargs assumes the correct type will be read. This is fragile because it cannot generally be enforced to be safe in the language and so relies on programmer discipline to get it right.

Example

```
int sum(...) {
    // ...
    while(/*...*/)
        result += va_arg(list, int); // BAD, assumes it will be passed ints
    // ...
}

sum(3, 2); // ok
sum(3.14159, 2.71828); // BAD, undefined

template<class ...Args>
auto sum(Args... args) { // GOOD, and much more flexible
    return (... + args); // note: C++17 "fold expression"
}
```

```
sum(3, 2); // ok: 5
sum(3.14159, 2.71828); // ok: ~5.85987
```

Note: Declaring a `...` parameter is sometimes useful for techniques that don't involve actual argument passing, notably to declare "take-anything" functions so as to disable "everything else" in an overload set or express a catchall case in a template metaprogram.

Enforcement

- Issue a diagnostic for using `va_list`, `va_start`, or `va_arg`. To fix: Use a variadic template parameter list instead.
- Issue a diagnostic for passing an argument to a vararg parameter of a function that does not offer an overload for a more specific type in the position of the vararg. To fix: Use a different function, or `[[suppress(types)]]`.

Bounds safety profile

This profile makes it easier to construct code that operates within the bounds of allocated blocks of memory. It does so by focusing on removing the primary sources of bounds violations: pointer arithmetic and array indexing. One of the core features of this profile is to restrict pointers to only refer to single objects, not arrays.

For the purposes of this document, bounds-safety is defined to be the property that a program does not use a variable to access memory outside of the range that was allocated and assigned to that variable. (Note that the safety is intended to be complete when combined also with Type safety and Lifetime safety, which cover other unsafe operations that allow bounds violations, such as type-unsafe casts that 'widen' pointers.)

The following are under consideration but not yet in the rules below, and may be better in other profiles:

- ???

An implementation of this profile shall recognize the following patterns in source code as non-conforming and issue a diagnostic.

Bounds.1: Don't use pointer arithmetic. Use `span` instead.

Reason Pointers should only refer to single objects, and pointer arithmetic is fragile and easy to get wrong. `span` is a bounds-checked, safe type for accessing arrays of data.

Example, bad

```
void f(int* p, int count)
{
    if (count < 2) return;

    int* q = p + 1; // BAD

    ptrdiff_t d;
    int n;
    d = (p - &n); // OK
    d = (q - p); // OK

    int n = *p++; // BAD

    if (count < 6) return;

    p[4] = 1; // BAD

    p[count - 1] = 2; // BAD

    use(&p[0], 3); // BAD
}
```

Example, good

```
void f(span<int> a) // BETTER: use span in the function declaration
{
    if (a.length() < 2) return;

    int n = *a++; // OK

    span<int> q = a + 1; // OK

    if (a.length() < 6) return;

    a[4] = 1; // OK

    a[count - 1] = 2; // OK

    use(a.data(), 3); // OK
}
```

Enforcement Issue a diagnostic for any arithmetic operation on an expression of pointer type that results in a value of pointer type.

Bounds.2: Only index into arrays using constant expressions.

Reason Dynamic accesses into arrays are difficult for both tools and humans to validate as safe. `span` is a bounds-checked, safe type for accessing arrays of data. `at()` is another alternative that ensures single accesses are bounds-checked. If iterators are needed to access an array, use the iterators from a `span` constructed over the array.

Example, bad

```
void f(array<int, 10> a, int pos)
{
    a[pos / 2] = 1; // BAD
    a[pos - 1] = 2; // BAD
    a[-1] = 3;    // BAD - no replacement, just don't do this
    a[10] = 4;    // BAD - no replacement, just don't do this
}
```

Example, good

```
// ALTERNATIVE A: Use a span

// A1: Change parameter type to use span
void f(span<int, 10> a, int pos)
{
    a[pos / 2] = 1; // OK
    a[pos - 1] = 2; // OK
}

// A2: Add local span and use that
void f(array<int, 10> arr, int pos)
{
    span<int> a = {arr, pos}
    a[pos / 2] = 1; // OK
    a[pos - 1] = 2; // OK
}

// ALTERNATIVE B: Use at() for access
void f(array<int, 10> a, int pos)
{
    at(a, pos / 2) = 1; // OK
```

```
    at(a, pos - 1) = 2; // OK
}
```

Example, bad

```
void f()
{
    int arr[COUNT];
    for (int i = 0; i < COUNT; ++i)
        arr[i] = i; // BAD, cannot use non-constant indexer
}
```

Example, good

```
// ALTERNATIVE A: Use a span
void f1()
{
    int arr[COUNT];
    span<int> av = arr;
    for (int i = 0; i < COUNT; ++i)
        av[i] = i;
}

// ALTERNATIVE B: Use at() for access
void f2()
{
    int arr[COUNT];
    for (int i = 0; i < COUNT; ++i)
        at(arr, i) = i;
}
```

Enforcement Issue a diagnostic for any indexing expression on an expression or variable of array type (either static array or `std::array`) where the indexer is not a compile-time constant expression.

Issue a diagnostic for any indexing expression on an expression or variable of array type (either static array or `std::array`) where the indexer is not a value between `0` or and the upper bound of the array.

Rewrite support: Tooling can offer rewrites of array accesses that involve dynamic index expressions to use `at()` instead:

```
static int a[10];
```

```
void f(int i, int j)
{
    a[i + j] = 12;      // BAD, could be rewritten as ...
    at(a, i + j) = 12;  // OK - bounds-checked
}
```

Bounds.3: No array-to-pointer decay.

Reason Pointers should not be used as arrays. `span` is a bounds-checked, safe alternative to using pointers to access arrays.

Example, bad

```
void g(int* p, size_t length);

void f()
{
    int a[5];
    g(a, 5);        // BAD
    g(&a[0], 1);    // OK
}
```

Example, good

```
void g(int* p, size_t length);
void g1(span<int> av); // BETTER: get g() changed.

void f()
{
    int a[5];
    span av = a;

    g(a.data(), a.length());   // OK, if you have no choice
    g1(a);                     // OK - no decay here, instead use implicit span ctor
}
```

Enforcement Issue a diagnostic for any expression that would rely on implicit conversion of an array type to a pointer type.

Bounds.4: Don't use standard library functions and types that are not bounds-checked.

Reason These functions all have bounds-safe overloads that take `span`. Standard types such as `vector` can be modified to perform bounds-checks under the bounds profile (in a compatible way, such as by adding contracts), or used with `at()`.

Example, bad

```
void f()
{
    array<int, 10> a, b;
    memset(a.data(), 0, 10);         // BAD, and contains a length error (length = 10 *
    memcmp(a.data(), b.data(), 10);  // BAD, and contains a length error (length = 10 *
}
```

Also, `std::array<>::fill()` or `std::fill()` or even an empty initializer are better candidate than `memset()`.

Example, good

```
void f()
{
    array<int, 10> a, b, c{};       // c is initialized to zero
    a.fill(0);
    fill(b.begin(), b.end(), 0);    // std::fill()
    fill(b, 0);                     // std::fill() + Ranges TS

    if ( a == b ) {
    }
}
```

Example If code is using an unmodified standard library, then there are still workarounds that enable use of `std::array` and `std::vector` in a bounds-safe manner. Code can call the `.at()` member function on each class, which will result in an `std::out_of_range` exception being thrown. Alternatively, code can call the `at()` free function, which will result in fail-fast (or a customized action) on a bounds violation.

```
void f(std::vector<int>& v, std::array<int, 12> a, int i)
{
    v[0] = a[0];        // BAD
    v.at(0) = a[0];     // OK (alternative 1)
    at(v, 0) = a[0];    // OK (alternative 2)

    v.at(0) = a[i];     // BAD
    v.at(0) = a.at(i)   // OK (alternative 1)
    v.at(0) = at(a, i); // OK (alternative 2)
}
```

Enforcement

- Issue a diagnostic for any call to a standard library function that is not bounds-checked. ??? insert link to a list of banned functions

TODO Notes:

- Impact on the standard library will require close coordination with WG21, if only to ensure compatibility even if never standardized.
- We are considering specifying bounds-safe overloads for stdlib (especially C stdlib) functions like `memcmp` and shipping them in the GSL.
- For existing stdlib functions and types like `vector` that are not fully bounds-checked, the goal is for these features to be bounds-checked when called from code with the bounds profile on, and unchecked when called from legacy code, possibly using contracts (concurrently being proposed by several WG21 members).

Lifetime safety profile

GSL: Guideline support library

The GSL is a small library of facilities designed to support this set of guidelines. Without these facilities, the guidelines would have to be far more restrictive on language details.

The Core Guidelines support library is defined in namespace `gsl` and the names may be aliases for standard library or other well-known library names. Using the (compile-time) indirection through the `gsl` namespace allows for experimentation and for local variants of the support facilities.

The support library facilities are designed to be extremely lightweight (zero-overhead) so that they impose no overhead compared to using conventional alternatives. Where desirable, they can be "instrumented" with additional functionality (e.g., checks) for tasks such as debugging.

These Guidelines assume a `variant` type, but this is not currently in GSL because the design is being actively refined in the standards committee.

GSL.view: Views

These types allow the user to distinguish between owning and non-owning pointers and between pointers to a single object and pointers to the first element of a sequence.

These "views" are never owners.

References are never owners.

The names are mostly ISO standard-library style (lower case and underscore):

- `T*` // The `T*` is not an owner, may be null; assumed to be pointing to a single element.
- `char*` // A C-style string (a zero-terminated array of characters); may be null.
- `const char*` // A C-style string; may be null.
- `T&` // The `T&` is not an owner and can never be a "null reference"; references are always bound to objects.

The "raw-pointer" notation (e.g. `int*`) is assumed to have its most common meaning; that is, a pointer points to an object, but does not own it. Owners should be converted to resource handles (e.g., `unique_ptr` or `vector<T>`) or marked `owner<T*>`

- `owner<T*>` // a `T*`that owns the object pointed/referred to; may be `nullptr`.
- `owner<T&>` // a `T&` that owns the object pointed/referred to.

`owner` is used to mark owning pointers in code that cannot be upgraded to use proper resource handles. Reasons for that include:

- Cost of conversion.
- The pointer is used with an ABI.
- The pointer is part of the implementation of a resource handle.

An `owner<T>` differs from a resource handle for a `T` by still requiring an explicit `delete`.

An `owner<T>` is assumed to refer to an object on the free store (heap).

If something is not supposed to be `nullptr`, say so:

- `not_null<T>` // `T` is usually a pointer type (e.g., `not_null<int*>` and `not_null<owner<Foo*>>`) that may not be `nullptr`. `T` can be any type for which `==nullptr` is meaningful.

- `span<T>` // [`p`:`p+n`), constructor from `{p, q}` and `{p, n}`; `T` is the pointer type

- `span_p<T>` // `{p, predicate}` [`p`:`q`) where `q` is the first element for which `predicate(*p)` is true

- `string_span` // `span<char>`

- `cstring_span` // `span<const char>`

A `span<T>` refers to zero or more mutable `T`s unless `T` is a `const` type.

"Pointer arithmetic" is best done within `span`s. A `char*` that points to something that is not a C-style string (e.g., a pointer into an input buffer) should be represented by a `span`. There is no really good way to say "pointer to a single `char`" (`string_span{p, 1}` can do that, and `T*` where `T` is a `char` in a template that has not been specialized for C-style strings).

- `zstring` // a `char*` supposed to be a C-style string; that is, a zero-terminated sequence of `char` or `null_ptr`
- `czstring` // a `const char*` supposed to be a C-style string; that is, a zero-terminated sequence of `const char` or `null_ptr`

Logically, those last two aliases are not needed, but we are not always logical, and they make the distinction between a pointer to one `char` and a pointer to a C-style string explicit. A sequence of characters that is not assumed to be zero-terminated should be a `char*`, rather than a `zstring`. French accent optional.

Use `not_null<zstring>` for C-style strings that cannot be `nullptr`. ??? Do we need a name for `not_null<zstring>`? or is its ugliness a feature?

GSL.owner: Ownership pointers

- `unique_ptr<T>` // unique ownership: `std::unique_ptr<T>`
- `shared_ptr<T>` // shared ownership: `std::shared_ptr<T>` (a counted pointer)
- `stack_array<T>` // A stack-allocated array. The number of elements are determined at construction and fixed thereafter. The elements are mutable unless `T` is a `const` type.
- `dyn_array<T>` // ??? needed ??? A heap-allocated array. The number of elements are determined at construction and fixed thereafter. The elements are mutable unless `T` is a `const` type. Basically a `span` that allocates and owns its elements.

GSL.assert: Assertions

- `Expects` // precondition assertion. Currently placed in function bodies. Later, should be moved to declarations. // `Expects(p)` terminates the program unless `p == true` // `Expect` in under control of some options (enforcement, error message, alternatives to terminate)
- `Ensures` // postcondition assertion. Currently placed in function bodies. Later, should be moved to declarations.

These assertions is currently macros (yuck!) pending standard commission decisions on contracts and assertion syntax.

GSL.util: Utilities

- `finally` // `finally(f)` makes a `final_action{f}` with a destructor that invokes `f`
- `narrow_cast` // `narrow_cast<T>(x)` is `static_cast<T>(x)`
- `narrow` // `narrow<T>(x)` is `static_cast<T>(x)` if `static_cast<T>(x) == x` or it throws `narrowing_error`

- `[[implicit]]` // "Marker" to put on single-argument constructors to explicitly make them non-explicit.
- `move_owner` // `p = move_owner(q)` means `p = q` but ???

GSL.concept: Concepts

These concepts (type predicates) are borrowed from Andrew Sutton's Origin library, the Range proposal, and the ISO WG21 Palo Alto TR. They are likely to be very similar to what will become part of the ISO C++ standard. The notation is that of the ISO WG21 Concepts TS (???ref???).

- `Range`
- `String` // ???
- `Number` // ???
- `Sortable`
- `Pointer` // A type with `*`, `->`, `==`, and default construction (default construction is assumed to set the singular "null" value) see smartptrconcepts
- `Unique_ptr` // A type that matches `Pointer`, has move (not copy), and matches the Lifetime profile criteria for a `unique` owner type see smartptrconcepts
- `Shared_ptr` // A type that matches `Pointer`, has copy, and matches the Lifetime profile criteria for a `shared` owner type see smartptrconcepts
- `EqualityComparable` // ???Must we suffer CaMelcAse???
- `Convertible`
- `Common`
- `Boolean`
- `Integral`
- `SignedIntegral`
- `SemiRegular`
- `Regular`
- `TotallyOrdered`
- `Function`
- `RegularFunction`
- `Predicate`
- `Relation`
- …

Smart pointer concepts

Described in Lifetimes paper.

NL: Naming and layout rules

Consistent naming and layout are helpful. If for no other reason because it minimizes "my style is better than your style" arguments. However, there are many, many, different styles around and people are passionate about them (pro and con). Also, most real-world projects includes code from many sources, so standardizing on a single style for all code is often impossible. We present a set of rules that you might use if you have no better ideas, but the real aim is consistency, rather than any particular rule set. IDEs and tools can help (as well as hinder).

Naming and layout rules:

- NL 1: Don't say in comments what can be clearly stated in code
- NL.2: State intent in comments
- NL.3: Keep comments crisp
- NL.4: Maintain a consistent indentation style
- NL.5: Don't encode type information in names
- NL.7: Make the length of a name roughly proportional to the length of its scope
- NL.8: Use a consistent naming style
- NL 9: Use `ALL_CAPS` for macro names only
- NL.10: Avoid CamelCase
- NL.15: Use spaces sparingly
- NL.16: Use a conventional class member declaration order
- NL.17: Use K&R-derived layout
- NL.18: Use C++-style declarator layout
- NL.25: Don't use `void` as an argument type

Most of these rules are aesthetic and programmers hold strong opinions. IDEs also tend to have defaults and a range of alternatives. These rules are suggested defaults to follow unless you have reasons not to.

More specific and detailed rules are easier to enforce.

NL.1: Don't say in comments what can be clearly stated in code

Reason Compilers do not read comments. Comments are less precise than code. Comments are not updated as consistently as code.

Example, bad

```
auto x = m*v1 + vv;    // multiply m with v1 and add the result to vv
```

Enforcement Build an AI program that interprets colloquial English text and see if what is said could be better expressed in C++.

NL.2: State intent in comments

Reason Code says what is done, not what is supposed to be done. Often intent can be stated more clearly and concisely than the implementation.

Example

```
void stable_sort(Sortable& c)
    // sort c in the order determined by <, keep equal elements (as defined by ==) in
{
    // ... quite a few lines of non-trivial code ...
}
```

Note If the comment and the code disagrees, both are likely to be wrong.

NL.3: Keep comments crisp

Reason Verbosity slows down understanding and makes the code harder to read by spreading it around in the source file.

Enforcement not possible.

NL.4: Maintain a consistent indentation style

Reason Readability. Avoidance of "silly mistakes."

Example, bad

```
int i;
for (i = 0; i < max; ++i); // bug waiting to happen
if (i == j)
    return i;
```

Enforcement Use a tool.

NL.5 Don't encode type information in names

Rationale: If names reflects type rather than functionality, it becomes hard to change the types used to provide that functionality. Names with types encoded are either verbose or cryptic. Hungarian notation is evil (at least in a strongly statically-typed language).

Example

???

Note Some styles distinguishes members from local variable, and/or from global variable.

```
struct S {
    int m_;
    S(int m) :m_{abs(m)} { }
};
```

This is not evil.

Note Some styles distinguishes types from non-types.

```
typename<typename T>
class Hash_tbl {   // maps string to T
    // ...
};

Hash_tbl<int> index;
```

This is not evil.

NL.7: Make the length of a name roughly proportional to the length of its scope

Rationale: ???

Example

???

Enforcement ???

NL.8: Use a consistent naming style

Rationale: Consistence in naming and naming style increases readability.

Note Where are many styles and when you use multiple libraries, you can't follow all their differences conventions. Choose a "house style", but leave "imported" libraries with their original style.

Example ISO Standard, use lower case only and digits, separate words with underscores:

- `int`
- `vector`
- `my_map`

Avoid double underscores `__`.

Example Stroustrup: ISO Standard, but with upper case used for your own types and concepts:

- `int`
- `vector`
- `My_map`

Example CamelCase: capitalize each word in a multi-word identifier:

- `int`
- `vector`
- `MyMap`
- `myMap`

Some conventions capitalize the first letter, some don't.

Note Try to be consistent in your use of acronyms and lengths of identifiers:

```
int mtbf {12};
int mean_time_between_failures {12}; // make up your mind
```

Enforcement Would be possible except for the use of libraries with varying conventions.

NL 9: Use `ALL_CAPS` for macro names only

Reason To avoid confusing macros from names that obeys scope and type rules.

Example

```
void f()
{
    const int SIZE{1000};  // Bad, use 'size' instead
    int v[SIZE];
}
```

Note This rule applies to non-macro symbolic constants:

```
enum bad { BAD, WORSE, HORRIBLE }; // BAD
```

Enforcement

- Flag macros with lower-case letters
- Flag `ALL_CAPS` non-macro names

NL.10: Avoid CamelCase

Reason The use of underscores to separate parts of a name is the original C and C++ style and used in the C++ standard library. If you prefer CamelCase, you have to choose among different flavors of camelCase.

Note This rule is a default to use only if you have a choice. Often, you don't have a choice and must follow an established style for consistency. The need for consistency beats personal taste.

Example Stroustrup: ISO Standard, but with upper case used for your own types and concepts:

- `int`
- `vector`
- `My_map`

Enforcement Impossible.

NL.15: Use spaces sparingly

Reason Too much space makes the text larger and distracts.

Example, bad

```
#include < map >

int main (int argc, char * argv [ ])
{
    // ...
}
```

Example

```
#include<map>

int main(int argc, char* argv[])
{
    // ...
}
```

Note Some IDEs have their own opinions and add distracting space.

Note We value well-placed whitespace as a significant help for readability. Just don't overdo it.

NL.16: Use a conventional class member declaration order

Reason A conventional order of members improves readability.

When declaring a class use the following order

- types: classes, enums, and aliases (`using`)
- constructors, assignments, destructor
- functions
- data

Use the `public` before `protected` before `private` order.

Private types and functions can be placed with private data.

Avoid multiple blocks of declarations of one access (e.g., `public`) dispersed among blocks of declarations with different access (e.g. `private`).

Example

```
???
```

Note The use of macros to declare groups of members often violates any ordering rules. However, macros obscures what is being expressed anyway.

Enforcement Flag departures from the suggested order. There will be a lot of old code that doesn't follow this rule.

NL.17: Use K&R-derived layout

Reason This is the original C and C++ layout. It preserves vertical space well. It distinguishes different language constructs (such as functions and classes) well.

Note In the context of C++, this style is often called "Stroustrup".

Example

```
struct Cable {
    int x;
    // ...
};

double foo(int x)
{
    if (0 < x) {
        // ...
    }

    switch (x) {
        case 0:
            // ...
            break;
        case amazing:
            // ...
            break;
        default:
            // ...
            break;
    }

    if (0 < x)
        ++x;

    if (x < 0)
        something();
    else
        something_else();

    return some_value;
}
```

Note a space between `if` and `(`

Note Use separate lines for each statement, the branches of an `if`, and the body of a `for`.

Note The `{` for a `class` and a `struct` in *not* on a separate line, but the `{` for a function is.

Note Capitalize the names of your user-defined types to distinguish them from standards-library types.

Note Do not capitalize function names.

Enforcement If you want enforcement, use an IDE to reformat.

NL.18: Use C++-style declarator layout

Reason The C-style layout emphasizes use in expressions and grammar, whereas the C++-style emphasizes types. The use in expressions argument doesn't hold for references.

Example

```
T& operator[](size_t);   // OK
T &operator[](size_t);   // just strange
T & operator[](size_t);   // undecided
```

Enforcement Impossible in the face of history.

NL.25: Don't use `void` as an argument type

Reason It's verbose and only needed where C compatibility matters.

Example

```
void f(void);   // bad

void g();       // better
```

Note

Even Dennis Ritchie deemed `void f(void)` an abomination. You can make an argument for that abomination in C when function prototypes were rare so that banning:

```
int f();
f(1, 2, "weird but valid C89");   // hope that f() is defined int f(a, b, c) char* c; {
```

would have caused major problems, but not in the 21st century and in C++.

FAQ: Answers to frequently asked questions

This section covers answers to frequently asked questions about these guidelines.

FAQ.1: What do these guidelines aim to achieve?

See the top of this page. This is an open source project to maintain modern authoritative guidelines for writing C++ code using the current C++ Standard (as of this writing, C++14). The guidelines are designed to be modern, machine-enforceable wherever possible, and open to contributions and forking so that organizations can easily incorporate them into their own corporate coding guidelines.

FAQ.2: When and where was this work first announced?

It was announced by Bjarne Stroustrup in his CppCon 2015 opening keynote, "Writing Good C++14". See also the accompanying isocpp.org blog post, and for the rationale of the type and memory safety guidelines see Herb Sutter's follow-up CppCon 2015 talk, "Writing Good C++14 … By Default".

FAQ.3: Who are the authors and maintainers of these guidelines?

The initial primary authors and maintainers are Bjarne Stroustrup and Herb Sutter, and the guidelines so far were developed with contributions from experts at CERN, Microsoft, Morgan Stanley, and several other organizations. At the time of their release, the guidelines are in a "0.6" state, and contributions are welcome. As Stroustrup said in his announcement: "We need help!"

FAQ.4: How can I contribute?

See CONTRIBUTING.md. We appreciate volunteer help!

FAQ.5: How can I become an editor/maintainer?

By contributing a lot first and having the consistent quality of your contributions recognized. See CONTRIBUTING.md. We appreciate volunteer help!

FAQ.6: Have these guidelines been approved by the ISO C++ standards committee? Do they represent the consensus of the committee?

No. These guidelines are outside the standard. They are intended to serve the standard, and be maintained as current guidelines about how to use the current Standard C++ effectively. We aim to keep them in sync with the standard as that is evolved by the committee.

FAQ.7: If these guidelines are not approved by the committee, why are they under `github.com/isocpp`?

Because `isocpp` is the Standard C++ Foundation; the committee's repositories are under github.com/*cplusplus*. Some neutral organization has to own the copyright and license to make it clear this is not being dominated by any one person or vendor. The natural entity is the Foundation, which exists to promote the use and up-to-date understanding of modern Standard C++ and the work of the committee. This follows the same pattern that isocpp.org did for the C++ FAQ, which was initially the work of Bjarne Stroustrup, Marshall Cline, and Herb Sutter and contributed to the open project in the same way.

FAQ.8: Will there be a C++98 version of these Guidelines? a C++11 version?

No. These guidelines are about how to best use Standard C++14 (and, if you have an implementation available, the Concepts Lite Technical Specification) and write code assuming you have a modern conforming compiler.

FAQ.9: Do these guidelines propose new language features?

No. These guidelines are about how to best use Standard C++14 + the Concepts Lite Technical Specification, and they limit themselves to recommending only those features.

FAQ.10: What version of Markdown do these guidelines use?

These coding standards are written using CommonMark, and `<a>` HTML anchors.

We are considering the following extensions from GitHub Flavored Markdown (GFM):

- fenced code blocks (consistently using indented vs. fenced is under discussion)
- tables (none yet but we'll likely need them, and this is a GFM extension)

Avoid other HTML tags and other extensions.

Note: We are not yet consistent with this style.

FAQ.50: What is the GSL (guideline support library)?

The GSL is the small set of types and aliases specified in these guidelines. As of this writing, their specification herein is too sparse; we plan to add a WG21-style interface specification to ensure that different implementations agree, and to propose as a contribution for possible standardization, subject as usual to whatever the committee decides to accept/improve/alter/reject.

FAQ.51: Is github.com/Microsoft/GSL the GSL?

No. That is just a first implementation contributed by Microsoft. Other implementations by other vendors are encouraged, as are forks of and contributions to that implementation. As of this writing one week into the public project, at least one GPLv3 open source implementation already exists. We plan to produce a WG21-style interface specification to ensure that different implementations agree.

FAQ.52: Why not supply an actual GSL implementation in/with these guidelines?

We are reluctant to bless one particular implementation because we do not want to make people think there is only one, and inadvertently stifle parallel implementations. And if these guidelines included an actual implementation, then whoever contributed it could be mistakenly seen as too influential. We prefer to follow the long-standing approach of the committee, namely to specify interfaces, not implementations. But at the same time we want at least one implementation available; we hope for many.

FAQ.53: Why weren't the GSL types proposed through Boost?

Because we want to use them immediately, and because they are temporary in that we want to retire them as soon as types that fill the same needs exist in the standard library.

FAQ.54: Has the GSL (guideline support library) been approved by the ISO C++ standards committee?

No. The GSL exists only to supply a few types and aliases that are not currently in the standard library. If the committee decides on standardized versions (of these or other types that fill the same need) then they can be removed from the GSL.

FAQ.55: If you're using the standard types where available, why is the GSL `string_span` different from the `string_view` in the Library Fundamentals 1 Technical Specification? Why not just use the committee-approved `string_view`?

Because `string_view` is still undergoing standardization, and is in a state for public review input to improve it. Types that appear in Technical Specifications (TSes) are not yet part of the International Standard (IS), and one reason they are put in TSes first is to gain experience with the feature before they are cast in a final form to become part of the standard. Some of the GSL authors are contributing what we have learned about `string_span` in the process of developing these guidelines, and a discussion of the differences between `string_view` and `string_span`, as a paper for the next ISO meeting for consideration along with all the other similar papers for the committee to consider as it decides on the final form of this feature.

FAQ.56: Is `owner` the same as the proposed `observer_ptr`?

No. `owner` owns, is an alias, and can be applied to any indirection type. The main intent of `observer_ptr` is to signify a *non*-owning pointer.

FAQ.57: Is `stack_array` the same as the standard `array`?

No. `stack_array` is guaranteed to be allocated on the stack. Although a `std::array` contains its storage directly inside itself, the `array` object can be put anywhere, including the heap.

FAQ.58: Is `dyn_array` the same as `vector` or the proposed `dynarray`?

No. `dyn_array` is not resizable, and is a safe way to refer to a heap-allocated fixed-size array. Unlike `vector`, it is intended to replace array-`new[]`. Unlike the `dynarray` that has been proposed in the committee, this does not anticipate compiler/language magic to somehow allocate it on the stack when it is a member of an object that is allocated on the stack; it simply refers to a "dynamic" or heap-based array.

FAQ.59: Is `Expects` the same as `assert`?

No. It is a placeholder for language support for contract preconditions.

FAQ.60: Is `Ensures` the same as `assert`?

No. It is a placeholder for language support for contract postconditions.

Appendix A: Libraries

This section lists recommended libraries, and explicitly recommends a few.

??? Suitable for the general guide? I think not ???

Appendix B: Modernizing code

Ideally, we follow all rules in all code. Realistically, we have to deal with a lot of old code:

- application code written before the guidelines were formulated or known
- libraries written to older/different standards
- code that we just haven't gotten around to modernizing

If we have a million lines of new code, the idea of "just changing it all at once" is typically unrealistic. Thus, we need a way of gradually modernizing a code base.

Upgrading older code to modern style can be a daunting task. Often, the old code is both a mess (hard to understand) and working correctly (for the current range of uses). Typically, the original programmer is not around and test cases incomplete. The fact that the code is a mess dramatically increases the effort needed to make any change and the risk of introducing errors. Often, messy old code runs unnecessarily slowly because it requires outdated compilers and cannot take advantage of modern hardware. In many cases, automated "modernizer"-style tool support would be required for major upgrade efforts.

The purpose of modernizing code is to simplify adding new functionality, to ease maintenance, and to increase performance (throughput or latency), and to better utilize modern hardware. Making code "look pretty" or "follow modern style" are not by themselves reasons for change. There are risks implied by every change and costs (including the cost of lost opportunities) implied by having an outdated code base. The cost reductions must outweigh the risks.

But how?

There is no one approach to modernizing code. How best to do it depends on the code, the pressure for updates, the backgrounds of the developers, and the available tool. Here are some (very general) ideas:

- The ideal is "just upgrade everything." That gives the most benefits for the shortest total time. In most circumstances, it is also impossible.
- We could convert a code base module for module, but any rules that affects interfaces (especially ABIs), such as use `span`, cannot be done on a per-module basis.
- We could convert code "bottom up" starting with the rules we estimate will give the greatest benefits and/or the least trouble in a given code base.
- We could start by focusing on the interfaces, e.g., make sure that no resources are lost and no pointer is misused. This would be a set of changes across the whole code base, but would most likely have huge benefits.

Whichever way you choose, please note that the most advantages come with the highest conformance to the guidelines. The guidelines are not a random set of unrelated rules where you can randomly pick and choose with an expectation of success.

We would dearly love to hear about experience and about tools used. Modernization can be much faster, simpler, and safer when supported with analysis tools and even code transformation tools.

Appendix C: Discussion

This section contains follow-up material on rules and sets of rules. In particular, here we present further rationale, longer examples, and discussions of alternatives.

Discussion: Define and initialize member variables in the order of member declaration

Member variables are always initialized in the order they are declared in the class definition, so write them in that order in the constructor initialization list. Writing them in a different order just makes the code confusing because it won't run in the order you see, and that can make it hard to see order-dependent bugs.

```
class Employee {
    string email, first, last;
public:
    Employee(const char* firstName, const char* lastName);
    // ...
};

Employee::Employee(const char* firstName, const char* lastName)
  : first(firstName),
    last(lastName),
    // BAD: first and last not yet constructed
    email(first + "." + last + "@acme.com")
{}
```

In this example, `email` will be constructed before `first` and `last` because it is declared first. That means its constructor will attempt to use `first` and `last` too soon – not just before they are set to the desired values, but before they are constructed at all.

If the class definition and the constructor body are in separate files, the long-distance influence that the order of member variable declarations has over the constructor's correctness will be even harder to spot.

References

[Cline99] §22.03-11, [Dewhurst03] §52-53, [Koenig97] §4, [Lakos96] §10.3.5, [Meyers97] §13, [Murray93] §2.1.3, [Sutter00] §47

Use of =, {}, and () as initializers

???

Discussion: Use a factory function if you need "virtual behavior" during initialization

If your design wants virtual dispatch into a derived class from a base class constructor or destructor for functions like `f` and `g`, you need other techniques, such as a post-constructor – a separate member function the caller must invoke to complete initialization, which can safely call `f` and `g` because in member functions virtual calls behave normally. Some techniques for this are shown in the References. Here's a non-exhaustive list of options:

- *Pass the buck:* Just document that user code must call the post-initialization function right after constructing an object.
- *Post-initialize lazily:* Do it during the first call of a member function. A Boolean flag in the base class tells whether or not post-construction has taken place yet.
- *Use virtual base class semantics:* Language rules dictate that the constructor most-derived class decides which base constructor will be invoked; you can use that to your advantage. (See [Taligent94].)
- *Use a factory function:* This way, you can easily force a mandatory invocation of a post-constructor function.

Here is an example of the last option:

```
class B {
public:
    B() { /* ... */ f(); /* ... */ }   // BAD: see Item 49.1

    virtual void f() = 0;

    // ...
};

class B {
protected:
    B() { /* ... */ }
    virtual void PostInitialize()    // called right after construction
        { /* ... */ f(); /* ... */ }   // GOOD: virtual dispatch is safe
public:
```

```
    virtual void f() = 0;

    template<class T>
    static shared_ptr<T> Create()    // interface for creating objects
    {
        auto p = make_shared<T>();
        p->PostInitialize();
        return p;
    }
};


class D : public B {                 // some derived class
public:
    void f() override { /* ...  */ };

protected:
    D() {}

    template<class T>
    friend shared_ptr<T> B::Create();
};

shared_ptr<D> p = D::Create<D>();    // creating a D object
```

This design requires the following discipline:

- Derived classes such as `D` must not expose a public constructor. Otherwise, `D`'s users could create `D` objects that don't invoke `PostInitialize`.
- Allocation is limited to `operator new`. `B` can, however, override `new` (see Items 45 and 46).
- `D` must define a constructor with the same parameters that `B` selected. Defining several overloads of `Create` can assuage this problem, however; and the overloads can even be templated on the argument types.

If the requirements above are met, the design guarantees that `PostInitialize` has been called for any fully constructed `B`-derived object. `PostInitialize` doesn't need to be virtual; it can, however, invoke virtual functions freely.

In summary, no post-construction technique is perfect. The worst techniques dodge the whole issue by simply asking the caller to invoke the post-constructor manually. Even the best require a different syntax for constructing objects (easy to check at compile time) and/or cooperation from derived class authors (impossible to check at compile time).

References: [Alexandrescu01] §3, [Boost], [Dewhurst03] §75, [Meyers97] §46, [Stroustrup00] §15.4.3, [Taligent94]

Discussion: Make base class destructors public and virtual, or protected and nonvirtual

Should destruction behave virtually? That is, should destruction through a pointer to a `base` class be allowed? If yes, then `base`'s destructor must be public in order to be callable, and virtual otherwise calling it results in undefined behavior. Otherwise, it should be protected so that only derived classes can invoke it in their own destructors, and nonvirtual since it doesn't need to behave virtually virtual.

Example The common case for a base class is that it's intended to have publicly derived classes, and so calling code is just about sure to use something like a `shared_ptr<base>`:

```
class base {
public:
    ~base();                   // BAD, not virtual
    virtual ~base();           // GOOD
    // ...
};

class derived : public base { /* ... */ };

{
    unique_ptr<base> pb = make_unique<derived>();
    // ...
} // ~pb invokes correct destructor only when ~base is virtual
```

In rarer cases, such as policy classes, the class is used as a base class for convenience, not for polymorphic behavior. It is recommended to make those destructors protected and nonvirtual:

```
class my_policy {
public:
    virtual ~my_policy();      // BAD, public and virtual
protected:
    ~my_policy();              // GOOD
    // ...
};

template<class Policy>
class customizable : Policy { /* ... */ }; // note: private inheritance
```

Note This simple guideline illustrates a subtle issue and reflects modern uses of inheritance and object-oriented design principles.

For a base class `Base`, calling code might try to destroy derived objects through pointers to `Base`, such as when using a `unique_ptr<Base>`. If `Base`'s destructor is public and nonvirtual (the default), it can be accidentally called on a pointer that actually points to a derived object, in which case the behavior of the attempted deletion is undefined. This state of affairs has led older coding standards to impose a blanket requirement that all base class destructors must be virtual. This is overkill (even if it is the common case); instead, the rule should be to make base class destructors virtual if and only if they are public.

To write a base class is to define an abstraction (see Items 35 through 37). Recall that for each member function participating in that abstraction, you need to decide:

- Whether it should behave virtually or not.
- Whether it should be publicly available to all callers using a pointer to Base or else be a hidden internal implementation detail.

As described in Item 39, for a normal member function, the choice is between allowing it to be called via a pointer to `Base` nonvirtually (but possibly with virtual behavior if it invokes virtual functions, such as in the NVI or Template Method patterns), virtually, or not at all. The NVI pattern is a technique to avoid public virtual functions.

Destruction can be viewed as just another operation, albeit with special semantics that make nonvirtual calls dangerous or wrong. For a base class destructor, therefore, the choice is between allowing it to be called via a pointer to `Base` virtually or not at all; "nonvirtually" is not an option. Hence, a base class destructor is virtual if it can be called (i.e., is public), and nonvirtual otherwise.

Note that the NVI pattern cannot be applied to the destructor because constructors and destructors cannot make deep virtual calls. (See Items 39 and 55.)

Corollary: When writing a base class, always write a destructor explicitly, because the implicitly generated one is public and nonvirtual. You can always `=default` the implementation if the default body is fine and you're just writing the function to give it the proper visibility and virtuality.

Exception Some component architectures (e.g., COM and CORBA) don't use a standard deletion mechanism, and foster different protocols for object disposal. Follow the local patterns and idioms, and adapt this guideline as appropriate.

Consider also this rare case:

- `B` is both a base class and a concrete class that can be instantiated by itself, and so the destructor must be public for `B` objects to be created and destroyed.
- Yet `B` also has no virtual functions and is not meant to be used polymorphically, and so although the destructor is public it does not need to be virtual.

Then, even though the destructor has to be public, there can be great pressure to not make it virtual because as the first virtual function it would incur all the run-time type overhead when the added functionality should never be needed.

In this rare case, you could make the destructor public and nonvirtual but clearly document that further-derived objects must not be used polymorphically as `B`'s. This is what was done with `std::unary_function`.

In general, however, avoid concrete base classes (see Item 35). For example, `unary_function` is a bundle-of-typedefs that was never intended to be instantiated standalone. It really makes no sense to give it a public destructor; a better design would be to follow this Item's advice and give it a protected nonvirtual destructor.

References: [C++CS] Item 50, [Cargill92] pp. 77-79, 207¸ [Cline99] §21.06, 21.12-13¸ [Henricson97] pp. 110-114¸ [Koenig97] Chapters 4, 11¸ [Meyers97] §14¸ [Stroustrup00] §12.4.2¸ [Sutter02] §27¸ [Sutter04] §18

Discussion: Usage of noexcept

???

Discussion: Destructors, deallocation, and swap must never fail

Never allow an error to be reported from a destructor, a resource deallocation function (e.g., `operator delete`), or a `swap` function using `throw`. It is nearly impossible to write useful code if these operations can fail, and even if something does go wrong it nearly never makes any sense to retry. Specifically, types whose destructors may throw an exception are flatly forbidden from use with the C++ standard library. Most destructors are now implicitly `noexcept` by default.

Example

```
class nefarious {
public:
    nefarious()  { /* code that could throw */ }   // ok
    ~nefarious() { /* code that could throw */ }   // BAD, should not throw
    // ...
};
```

1. `nefarious` objects are hard to use safely even as local variables:

```
    void test(string& s)
    {
        nefarious n;          // trouble brewing
```

```
    string copy = s;      // copy the string
} // destroy copy and then n
```

```
Here, copying `s` could throw, and if that throws and if `n`'s destructor then also thro
```

2. Classes with `nefarious` members or bases are also hard to use safely, because their destructors must invoke `nefarious`' destructor, and are similarly poisoned by its poor behavior:

```
class innocent_bystander {
    nefarious member;     // oops, poisons the enclosing class's destructor
    // ...
};

void test(string& s)
{
    innocent_bystander i; // more trouble brewing
    string copy2 = s;      // copy the string
} // destroy copy and then i
```

```
Here, if constructing `copy2` throws, we have the same problem because `i`'s destructor
```

3. You can't reliably create global or static `nefarious` objects either:

```
static nefarious n;       // oops, any destructor exception can't be caught
```

4. You can't reliably create arrays of `nefarious`:

```
void test()
{
    std::array<nefarious, 10> arr; // this line can std::terminate(!)
}
```

```
The behavior of arrays is undefined in the presence of destructors that throw because th
```

5. You can't use `Nefarious` objects in standard containers:

```
std::vector<nefarious> vec(10);   // this line can std::terminate()
```

```
The standard library forbids all destructors used with it from throwing. You can't store
```

Note These are key functions that must not fail because they are necessary for the two key operations in transactional programming: to back out work if problems are encountered during processing, and to commit work if no problems occur. If there's no way to safely back out using no-fail operations, then no-fail rollback is impossible to implement. If there's no way to safely commit state changes using a no-fail operation (notably, but not limited to, `swap`), then no-fail commit is impossible to implement.

Consider the following advice and requirements found in the C++ Standard:

> If a destructor called during stack unwinding exits with an exception, terminate is called (15.5.1). So destructors should generally catch exceptions and not let them propagate out of the destructor. –[C++03] §15.2(3)
>
> No destructor operation defined in the C++ Standard Library (including the destructor of any type that is used to instantiate a standard library template) will throw an exception. –[C++03] §17.4.4.8(3)

Deallocation functions, including specifically overloaded `operator delete` and `operator delete[]`, fall into the same category, because they too are used during cleanup in general, and during exception handling in particular, to back out of partial work that needs to be undone. Besides destructors and deallocation functions, common error-safety techniques rely also on `swap` operations never failing–in this case, not because they are used to implement a guaranteed rollback, but because they are used to implement a guaranteed commit. For example, here is an idiomatic implementation of `operator=` for a type `T` that performs copy construction followed by a call to a no-fail `swap`:

```
T& T::operator=(const T& other) {
    auto temp = other;
    swap(temp);
}
```

(See also Item 56. ???)

Fortunately, when releasing a resource, the scope for failure is definitely smaller. If using exceptions as the error reporting mechanism, make sure such functions handle all exceptions and other errors that their internal processing might generate. (For exceptions, simply wrap everything sensitive that your destructor does in a `try/catch(...)` block.) This is particularly important because a destructor might be called in a crisis situation, such as failure to allocate a system resource (e.g., memory, files, locks, ports, windows, or other system objects).

When using exceptions as your error handling mechanism, always document this behavior by declaring these functions `noexcept`. (See Item 75.)

References: [C++CS] Item 51; [C++03] §15.2(3), §17.4.4.8(3)¸ [Meyers96] §11¸ [Stroustrup00] §14.4.7, §E.2-4¸ [Sutter00] §8, §16¸ [Sutter02] §18-19

Define Copy, move, and destroy consistently

Reason ???

Note If you define a copy constructor, you must also define a copy assignment operator.

Note If you define a move constructor, you must also define a move assignment operator.

Example

```
class x {
    // ...
public:
    x(const x&) { /* stuff */ }

    // BAD: failed to also define a copy assignment operator

    x(x&&) { /* stuff */ }

    // BAD: failed to also define a move assignment operator
};

x x1;
x x2 = x1; // ok
x2 = x1;   // pitfall: either fails to compile, or does something suspicious
```

If you define a destructor, you should not use the compiler-generated copy or move operation; you probably need to define or suppress copy and/or move.

```
class X {
    HANDLE hnd;
    // ...
public:
    ~X() { /* custom stuff, such as closing hnd */ }
    // suspicious: no mention of copying or moving -- what happens to hnd?
};

X x1;
X x2 = x1; // pitfall: either fails to compile, or does something suspicious
x2 = x1;   // pitfall: either fails to compile, or does something suspicious
```

If you define copying, and any base or member has a type that defines a move operation, you should also define a move operation.

```
class x {
    string s; // defines more efficient move operations
    // ... other data members ...
public:
    x(const x&) { /* stuff */ }
    x& operator=(const x&) { /* stuff */ }

    //    BAD: failed to also define a move construction and move assignment
    //   (why wasn't the custom "stuff" repeated here?)
};

x test()
{
    x local;
    // ...
    return local;  // pitfall: will be inefficient and/or do the wrong thing
}
```

If you define any of the copy constructor, copy assignment operator, or destructor, you probably should define the others.

Note If you need to define any of these five functions, it means you need it to do more than its default behavior–and the five are asymmetrically interrelated. Here's how:

- If you write/disable either of the copy constructor or the copy assignment operator, you probably need to do the same for the other: If one does "special" work, probably so should the other because the two functions should have similar effects. (See Item 53, which expands on this point in isolation.)
- If you explicitly write the copying functions, you probably need to write the destructor: If the "special" work in the copy constructor is to allocate or duplicate some resource (e.g., memory, file, socket), you need to deallocate it in the destructor.
- If you explicitly write the destructor, you probably need to explicitly write or disable copying: If you have to write a nontrivial destructor, it's often because you need to manually release a resource that the object held. If so, it is likely that those resources require careful duplication, and then you need to pay attention to the way objects are copied and assigned, or disable copying completely.

In many cases, holding properly encapsulated resources using RAII "owning" objects can eliminate the need to write these operations yourself. (See Item 13.)

Prefer compiler-generated (including `=default`) special members; only these can be classified as "trivial", and at least one major standard library vendor heavily optimizes for classes having trivial special members. This is likely to become common practice.

Exceptions: When any of the special functions are declared only to make them nonpublic or virtual, but without special semantics, it doesn't imply that the others are needed. In rare cases, classes that have members of strange types (such as reference members) are an exception because they have peculiar copy semantics. In a class holding a reference, you likely need to write the copy constructor and the assignment operator, but the default destructor already does the right thing. (Note that using a reference member is almost always wrong.)

References: [C++CS] Item 52; [Cline99] §30.01-14¸ [Koenig97] §4¸ [Stroustrup00] §5.5, §10.4¸ [SuttHysl04b]

Resource management rule summary:

- Provide strong resource safety; that is, never leak anything that you think of as a resource
- Never throw while holding a resource not owned by a handle
- A "raw" pointer or reference is never a resource handle
- Never let a pointer outlive the object it points to
- Use templates to express containers (and other resource handles)
- Return containers by value (relying on move or copy elision for efficiency)
- If a class is a resource handle, it needs a constructor, a destructor, and copy and/or move operations
- If a class is a container, give it an initializer-list constructor

Provide strong resource safety; that is, never leak anything that you think of as a resource

Reason Prevent leaks. Leaks can lead to performance degradation, mysterious error, system crashes, and security violations.

Alternative formulation: Have every resource represented as an object of some class managing its lifetime.

Example

```
template<class T>
class Vector {
// ...
private:
    T* elem;   // sz elements on the free store, owned by the class object
    int sz;
};
```

This class is a resource handle. It manages the lifetime of the `T`s. To do so, `Vector` must define or delete the set of special operations (constructors, a destructor, etc.).

Example

```
??? "odd" non-memory resource ???
```

Enforcement The basic technique for preventing leaks is to have every resource owned by a resource handle with a suitable destructor. A checker can find "naked `news`". Given a list of C-style allocation functions (e.g., `fopen()`), a checker can also find uses that are not managed by a resource handle. In general, "naked pointers" can be viewed with suspicion, flagged, and/or analyzed. A complete list of resources cannot be generated without human input (the definition of "a resource" is necessarily too general), but a tool can be "parameterized" with a resource list.

Never throw while holding a resource not owned by a handle

Reason That would be a leak.

Example

```
void f(int i)
{
    FILE* f = fopen("a file", "r");
    ifstream is { "another file" };
    // ...
    if (i == 0) return;
    // ...
    fclose(f);
}
```

If `i == 0` the file handle for `a file` is leaked. On the other hand, the `ifstream` for `another file` will correctly close its file (upon destruction). If you must use an explicit pointer, rather than a resource handle with specific semantics, use a `unique_ptr` or a `shared_ptr` with a custom deleter:

```
void f(int i)
{
    unique_ptr<FILE, int(*)(FILE*)> f(fopen("a file", "r"), fclose);
    // ...
    if (i == 0) return;
    // ...
}
```

Better:

```
void f(int i)
{
    ifstream input {"a file"};
    // ...
    if (i == 0) return;
    // ...
}
```

Enforcement A checker must consider all “naked pointers” suspicious. A checker probably must rely on a human-provided list of resources. For starters, we know about the standard-library containers, `string`, and smart pointers. The use of `span` and `string_span` should help a lot (they are not resource handles).

A “raw” pointer or reference is never a resource handle

Reason To be able to distinguish owners from views.

Note This is independent of how you “spell” pointer: `T*`, `T&`, `Ptr<T>` and `Range<T>` are not owners.

Never let a pointer outlive the object it points to

Reason To avoid extremely hard-to-find errors. Dereferencing such a pointer is undefined behavior and could lead to violations of the type system.

Example

```
string* bad()   // really bad
{
    vector<string> v = { "this", "will", "cause" "trouble" };
    return &v[0];   // leaking a pointer into a destroyed member of a destroyed object
}

void use()
{
    string* p = bad();
    vector<int> xx = {7, 8, 9};
    string x = *p;   // undefined behavior: x may not be "this"
    *p = "Evil!";    // undefined behavior: we don't know what (if anything) is alloca
}
```

The strings of v are destroyed upon exit from bad() and so is v itself. The returned pointer points to unallocated memory on the free store. This memory (pointed into by p) may have been reallocated by the time *p is executed. There may be no string to read and a write through p could easily corrupt objects of unrelated types.

Enforcement Most compilers already warn about simple cases and has the information to do more. Consider any pointer returned from a function suspect. Use containers, resource handles, and views (e.g., span known not to be resource handles) to lower the number of cases to be examined. For starters, consider every class with a destructor as resource handle.

Use templates to express containers (and other resource handles)

Reason To provide statically type-safe manipulation of elements.

Example

```
template<typename T> class Vector {
    // ...
    T* elem;   // point to sz elements of type T
    int sz;
};
```

Return containers by value (relying on move or copy elision for efficiency)

Reason To simplify code and eliminate a need for explicit memory management. To bring an object into a surrounding scope, thereby extending its lifetime. See also F.20, the general item about "out" output values.

Example

```
vector<int> get_large_vector()
{
    return ...;
}

auto v = get_large_vector(); //  return by value is ok, most modern compilers will do c
```

Exceptions See the Exceptions in F.20.

Enforcement Check for pointers and references returned from functions and see if they are assigned to resource handles (e.g., to a unique_ptr).

If a class is a resource handle, it needs a constructor, a destructor, and copy and/or move operations

Reason To provide complete control of the lifetime of the resource. To provide a coherent set of operations on the resource.

Example

```
??? Messing with pointers
```

Note If all members are resource handles, rely on the default special operations where possible.

```
template<typename T> struct Named {
    string name;
    T value;
};
```

Now `Named` has a default constructor, a destructor, and efficient copy and move operations, provided `T` has.

Enforcement In general, a tool cannot know if a class is a resource handle. However, if a class has some of the default operations, it should have all, and if a class has a member that is a resource handle, it should be considered as resource handle.

If a class is a container, give it an initializer-list constructor

Reason It is common to need an initial set of elements.

Example

```
template<typename T> class Vector {
public:
    vector<std::initializer_list<T>>;
    // ...
};

Vector<string> vs = { "Nygaard", "Ritchie" };
```

Enforcement When is a class a container? ???

Glossary

A relatively informal definition of terms used in the guidelines (based of the glossary in Programming: Principles and Practice using C++)

- *abstract class*: a class that cannot be directly used to create objects; often used to define an interface to derived classes. A class is made abstract by having a pure virtual function or a protected constructor.
- *abstraction*: a description of something that selectively and deliberately ignores (hides) details (e.g., implementation details); selective ignorance.
- *address*: a value that allows us to find an object in a computer's memory.
- *algorithm*: a procedure or formula for solving a problem; a finite series of computational steps to produce a result.
- *alias*: an alternative way of referring to an object; often a name, pointer, or reference.
- *application*: a program or a collection of programs that is considered an entity by its users.
- *approximation*: something (e.g., a value or a design) that is close to the perfect or ideal (value or design). Often an approximation is a result of trade-offs among ideals.
- *argument*: a value passed to a function or a template, in which it is accessed through a parameter.
- *array*: a homogeneous sequence of elements, usually numbered, e.g., [0:max).
- *assertion*: a statement inserted into a program to state (assert) that something must always be true at this point in the program.
- *base class*: a class used as the base of a class hierarchy. Typically a base class has one or more virtual functions.
- *bit*: the basic unit of information in a computer. A bit can have the value 0 or the value 1.
- *bug*: an error in a program.
- *byte*: the basic unit of addressing in most computers. Typically, a byte holds 8 bits.
- *class*: a user-defined type that may contain data members, function members, and member types.
- *code*: a program or a part of a program; ambiguously used for both source code and object code.
- *compiler*: a program that turns source code into object code.
- *complexity*: a hard-to-precisely-define notion or measure of the difficulty of constructing a solution to a problem or of the solution itself. Sometimes complexity is used to (simply) mean an estimate of the number of operations needed to execute an algorithm.
- *computation*: the execution of some code, usually taking some input and producing some output.
- *concept*: (1) a notion, and idea; (2) a set of requirements, usually for a template argument.
- *concrete class*: class for which objects can be created.
- *constant*: a value that cannot be changed (in a given scope); not mutable.

- *constructor*: an operation that initializes ("constructs") an object. Typically a constructor establishes an invariant and often acquires resources needed for an object to be used (which are then typically released by a destructor).
- *container*: an object that holds elements (other objects).
- *copy*: an operation that makes two object have values that compare equal. See also move.
- *correctness*: a program or a piece of a program is correct if it meets its specification. Unfortunately, a specification can be incomplete or inconsistent, or can fail to meet users' reasonable expectations. Thus, to produce acceptable code, we sometimes have to do more than just follow the formal specification.
- *cost*: the expense (e.g., in programmer time, run time, or space) of producing a program or of executing it. Ideally, cost should be a function of complexity.
- *customization point*: ???
- *data*: values used in a computation.
- *debugging*: the act of searching for and removing errors from a program; usually far less systematic than testing.
- *declaration*: the specification of a name with its type in a program.
- *definition*: a declaration of an entity that supplies all information necessary to complete a program using the entity. Simplified definition: a declaration that allocates memory.
- *derived class*: a class derived from one or more base classes.
- *design*: an overall description of how a piece of software should operate to meet its specification.
- *destructor*: an operation that is implicitly invoked (called) when an object is destroyed (e.g., at the end of a scope). Often, it releases resources.
- *encapsulation*: protecting something meant to be private (e.g., implementation details) from unauthorized access.
- *error*: a mismatch between reasonable expectations of program behavior (often expressed as a requirement or a users' guide) and what a program actually does.
- *executable*: a program ready to be run (executed) on a computer.
- *feature creep*: a tendency to add excess functionality to a program "just in case."
- *file*: a container of permanent information in a computer.
- *floating-point number*: a computer's approximation of a real number, such as 7.93 and 10.78e–3.
- *function*: a named unit of code that can be invoked (called) from different parts of a program; a logical unit of computation.
- *generic programming*: a style of programming focused on the design and efficient implementation of algorithms. A generic algorithm will work for all argument types that meet its requirements. In C++, generic programming typically uses templates.
- *Global variable*: Technically, a named object in namespace scope
- *handle*: a class that allows access to another through a member pointer or reference. See also resource, copy, move.
- *header*: a file containing declarations used to share interfaces between parts of a program.

- *hiding*: the act of preventing a piece of information from being directly seen or accessed. For example, a name from a nested (inner) scope can prevent that same name from an outer (enclosing) scope from being directly used.
- *ideal*: the perfect version of something we are striving for. Usually we have to make trade-offs and settle for an approximation.
- *implementation*: (1) the act of writing and testing code; (2) the code that implements a program.
- *infinite loop*: a loop where the termination condition never becomes true. See iteration.
- *infinite recursion*: a recursion that doesn't end until the machine runs out of memory to hold the calls. In reality, such recursion is never infinite but is terminated by some hardware error.
- *information hiding*: the act of separating interface and implementation, thus hiding implementation details not meant for the user's attention and providing an abstraction.
- *initialize*: giving an object its first (initial) value.
- *input*: values used by a computation (e.g., function arguments and characters typed on a keyboard).
- *integer*: a whole number, such as 42 and –99.
- *interface*: a declaration or a set of declarations specifying how a piece of code (such as a function or a class) can be called.
- *invariant*: something that must be always true at a given point (or points) of a program; typically used to describe the state (set of values) of an object or the state of a loop before entry into the repeated statement.
- *iteration*: the act of repeatedly executing a piece of code; see recursion.
- *iterator*: an object that identifies an element of a sequence.
- *library*: a collection of types, functions, classes, etc. implementing a set of facilities (abstractions) meant to be potentially used as part of more that one program.
- *lifetime*: the time from the initialization of an object until it becomes unusable (goes out of scope, is deleted, or the program terminates).
- *linker*: a program that combines object code files and libraries into an executable program.
- *literal*: a notation that directly specifies a value, such as 12 specifying the integer value "twelve."
- *loop*: a piece of code executed repeatedly; in C++, typically a for-statement or a while-statement.
- *move*: an operation that transfers a value from one object to another leaving behind a value representing "empty." See also copy.
- *mutable*: changeable; the opposite of immutable, constant, and variable.
- *object*: (1) an initialized region of memory of a known type which holds a value of that type; (2) a region of memory.
- *object code*: output from a compiler intended as input for a linker (for the linker to produce executable code).
- *object file*: a file containing object code.
- *object-oriented programming*: (OOP) a style of programming focused on the design and

use of classes and class hierarchies.

- *operation*: something that can perform some action, such as a function and an operator.
- *output*: values produced by a computation (e.g., a function result or lines of characters written on a screen).
- *overflow*: producing a value that cannot be stored in its intended target.
- *overload*: defining two functions or operators with the same name but different argument (operand) types.
- *override*: defining a function in a derived class with the same name and argument types as a virtual function in the base class, thus making the function callable through the interface defined by the base class.
- *owner*: an object responsible for releasing a resource.
- *paradigm*: a somewhat pretentious term for design or programming style; often used with the (erroneous) implication that there exists a paradigm that is superior to all others.
- *parameter*: a declaration of an explicit input to a function or a template. When called, a function can access the arguments passed through the names of its parameters.
- *pointer*: (1) a value used to identify a typed object in memory; (2) a variable holding such a value.
- *post-condition*: a condition that must hold upon exit from a piece of code, such as a function or a loop.
- *pre-condition*: a condition that must hold upon entry into a piece of code, such as a function or a loop.
- *program*: code (possibly with associated data) that is sufficiently complete to be executed by a computer.
- *programming*: the art of expressing solutions to problems as code.
- *programming language*: a language for expressing programs.
- *pseudo code*: a description of a computation written in an informal notation rather than a programming language.
- *pure virtual function*: a virtual function that must be overridden in a derived class.
- *RAII*: ("Resource Acquisition Is Initialization") a basic technique for resource management based on scopes.
- *range*: a sequence of values that can be described by a start point and an end point. For example, [0:5) means the values 0, 1, 2, 3, and 4.
- *regular expression*: a notation for patterns in character strings.
- *recursion*: the act of a function calling itself; see also iteration.
- *reference*: (1) a value describing the location of a typed value in memory; (2) a variable holding such a value.
- *requirement*: (1) a description of the desired behavior of a program or part of a program; (2) a description of the assumptions a function or template makes of its arguments.
- *resource*: something that is acquired and must later be released, such as a file handle, a lock, or memory. See also handle, owner.
- *rounding*: conversion of a value to the mathematically nearest value of a less precise type.

- *RTTI*: Run-Time Type Information. ???
- *scope*: the region of program text (source code) in which a name can be referred to.
- *sequence*: elements that can be visited in a linear order.
- *software*: a collection of pieces of code and associated data; often used interchangeably with program.
- *source code*: code as produced by a programmer and (in principle) readable by other programmers.
- *source file*: a file containing source code.
- *specification*: a description of what a piece of code should do.
- *standard*: an officially agreed upon definition of something, such as a programming language.
- *state*: a set of values.
- *STL*: the containers, iterators, and algorithms part of the standard library.
- *string*: a sequence of characters.
- *style*: a set of techniques for programming leading to a consistent use of language features; sometimes used in a very restricted sense to refer just to low-level rules for naming and appearance of code.
- *subtype*: derived type; a type that has all the properties of a type and possibly more.
- *supertype*: base type; a type that has a subset of the properties of a type.
- *system*: (1) a program or a set of programs for performing a task on a computer; (2) a shorthand for “operating system”, that is, the fundamental execution environment and tools for a computer.
- *template*: a class or a function parameterized by one or more types or (compile-time) values; the basic C++ language construct supporting generic programming.
- *testing*: a systematic search for errors in a program.
- *trade-off*: the result of balancing several design and implementation criteria.
- *truncation*: loss of information in a conversion from a type into another that cannot exactly represent the value to be converted.
- *type*: something that defines a set of possible values and a set of operations for an object.
- *uninitialized*: the (undefined) state of an object before it is initialized.
- *unit*: (1) a standard measure that gives meaning to a value (e.g., km for a distance); (2) a distinguished (e.g., named) part of a larger whole.
- *use case*: a specific (typically simple) use of a program meant to test its functionality and demonstrate its purpose.
- *value*: a set of bits in memory interpreted according to a type.
- *variable*: a named object of a given type; contains a value unless uninitialized.
- *virtual function*: a member function that can be overridden in a derived class.
- *word*: a basic unit of memory in a computer, often the unit used to hold an integer.

To-do: Unclassified proto-rules

This is our to-do list. Eventually, the entries will become rules or parts of rules. Alternatively, we will decide that no change is needed and delete the entry.

- No long-distance friendship
- Should physical design (what's in a file) and large-scale design (libraries, groups of libraries) be addressed?
- Namespaces
- How granular should namespaces be? All classes/functions designed to work together and released together (as defined in Sutter/Alexandrescu) or something narrower or wider?
- Should there be inline namespaces (à la `std::literals::*_literals`)?
- Avoid implicit conversions
- Const member functions should be thread safe "¦ aka, but I don't really change the variable, just assign it a value the first time it's called"¦ argh
- Always initialize variables, use initialization lists for member variables.
- Anyone writing a public interface which takes or returns void* should have their toes set on fire. That one has been a personal favorite of mine for a number of years. :)
- Use `const`-ness wherever possible: member functions, variables and (yippee) `const_iterators`
- Use `auto`
- `(size)` vs. `{initializers}` vs. `{Extent{size}}`
- Don't overabstract
- Never pass a pointer down the call stack
- falling through a function bottom
- Should there be guidelines to choose between polymorphisms? YES. classic (virtual functions, reference semantics) vs. Sean Parent style (value semantics, type-erased, kind of like `std::function`) vs. CRTP/static? YES Perhaps even vs. tag dispatch?
- Speaking of virtual functions, should non-virtual interface be promoted? YES. (public non-virtual `foo()` calling private/protected `do_foo()`)? Not a new thing, seeing as locales/streams use it, but it seems to be under-emphasized.
- should virtual calls be banned from ctors/dtors in your guidelines? YES. A lot of people ban them, even though I think it's a big strength of C++ that they are ??? -preserving (D disappointed me so much when it went the Java way). WHAT WOULD BE A GOOD EXAMPLE?
- Speaking of lambdas, what would weigh in on the decision between lambdas and (local?) classes in algorithm calls and other callback scenarios?
- And speaking of `std::bind`, Stephen T. Lavavej criticizes it so much I'm starting to wonder if it is indeed going to fade away in future. Should lambdas be recommended instead?
- What to do with leaks out of temporaries? : `p = (s1 + s2).c_str();`
- pointer/iterator invalidation leading to dangling pointers:

“‘cpp void bad() { int* p = new int[700]; int* q = &p[7]; delete p;

```
    vector<int> v(700);
    int* q2 = &v[7];
    v.resize(900);

    // ... use q and q2 ...
}
```

“‘

- LSP
- private inheritance vs/and membership
- avoid static class members variables (race conditions, almost-global variables)
- Use RAII lock guards (`lock_guard`, `unique_lock`, `shared_lock`), never call `mutex.lock` and `mutex.unlock` directly (RAII)
- Prefer non-recursive locks (often used to work around bad reasoning, overhead)
- Join your threads! (because of `std::terminate` in destructor if not joined or detached … is there a good reason to detach threads?) – ??? could support library provide a RAII wrapper for `std::thread`?
- If two or more mutexes must be acquired at the same time, use `std::lock` (or another deadlock avoidance algorithm?)
- When using a `condition_variable`, always protect the condition by a mutex (atomic bool whose value is set outside of the mutex is wrong!), and use the same mutex for the condition variable itself.
- Never use `atomic_compare_exchange_strong` with `std::atomic<user-defined-struct>` (differences in padding matter, while `compare_exchange_weak` in a loop converges to stable padding)
- individual `shared_future` objects are not thread-safe: two threads cannot wait on the same `shared_future` object (they can wait on copies of a `shared_future` that refer to the same shared state)
- individual `shared_ptr` objects are not thread-safe: different threads can call non-`const` member functions on *different* `shared_ptr`s that refer to the same shared object, but one thread cannot call a non-`const` member function of a `shared_ptr` object while another thread accesses that same `shared_ptr` object (if you need that, consider `atomic_shared_ptr` instead)
- rules for arithmetic

Bibliography

- [Alexandrescu01]: A. Alexandrescu. Modern C++ Design (Addison-Wesley, 2001).
- [C++03]: ISO/IEC 14882:2003(E), Programming Languages — C++ (updated ISO and ANSI C++ Standard including the contents of (C++98) plus errata corrections).
- [C++CS]:
- [Cargill92]: T. Cargill. C++ Programming Style (Addison-Wesley, 1992).
- [Cline99]: M. Cline, G. Lomow, and M. Girou. C++ FAQs (2ndEdition) (Addison-Wesley, 1999).
- [Dewhurst03]: S. Dewhurst. C++ Gotchas (Addison-Wesley, 2003).
- [Henricson97]: M. Henricson and E. Nyquist. Industrial Strength C++ (Prentice Hall, 1997).
- [Koenig97]: A. Koenig and B. Moo. Ruminations on C++ (Addison-Wesley, 1997).
- [Lakos96]: J. Lakos. Large-Scale C++ Software Design (Addison-Wesley, 1996).
- [Meyers96]: S. Meyers. More Effective C++ (Addison-Wesley, 1996).
- [Meyers97]: S. Meyers. Effective C++ (2nd Edition) (Addison-Wesley, 1997).
- [Meyers15]: S. Meyers. Effective Modern C++ (O'Reilly, 2015).
- [Murray93]: R. Murray. C++ Strategies and Tactics (Addison-Wesley, 1993).
- [Stroustrup00]: B. Stroustrup. The C++ Programming Language (Special 3rdEdition) (Addison-Wesley, 2000).
- [Stroustrup05]: B. Stroustrup. A rationale for semantically enhanced library languages.
- [Stroustrup13]: B. Stroustrup. The C++ Programming Language (4th Edition). Addison Wesley 2013.
- [Stroustrup14]: B. Stroustrup. A Tour of C++. Addison Wesley 2014.
- [SuttHysl04b]: H. Sutter and J. Hyslop. "Collecting Shared Objects" (C/C++ Users Journal, 22(8), August 2004).
- [SuttAlex05]: H. Sutter and A. Alexandrescu. C++ Coding Standards. Addison-Wesley 2005.
- [Sutter00]: H. Sutter. Exceptional C++ (Addison-Wesley, 2000).
- [Sutter02]: H. Sutter. More Exceptional C++ (Addison-Wesley, 2002).
- [Sutter04]: H. Sutter. Exceptional C++ Style (Addison-Wesley, 2004).
- [Taligent94]: Taligent's Guide to Designing Programs (Addison-Wesley, 1994).