Design Style

Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it.

—Alan Perlis

But I also knew, and forgot, Hoare's dictum that
premature optimization is the root of all evil in programming.

—Donald Knuth,
The Errors ofTeX [Knuth89]

It's difficult to fully separate Design Style and Coding Style. We have tried to leave to the next section those Items that generally crop up when actually writing code.

This section focuses on principles and practices that apply more broadly than just to a particular class or function. A classic case in point is the balance among simplicity and clarity (Item 6), avoiding premature optimization (Item 8), and avoiding premature pessimization (Item 9). Those three Items apply, not just at the function-coding level, but to the larger areas of class and module design tradeoffs and to far-reaching application architecture decisions. (They also apply to all programmers. If you think otherwise, please reread the above Knuth quote and note its citation.)

Following that, many of the other Items in this and the following section deal with aspects of dependency management—a cornerstone of software engineering and a recurring theme throughout the book. Stop and think of some random good software engineering technique—*any* good technique. Whichever one you picked, in one way or another it will be about reducing dependencies. Inheritance? Make code written to use the base class less dependent on the actual derived class. Minimize global variables? Reduce long-distance dependencies through widely visible data. Abstraction? Eliminate dependencies between code that manipulates concepts and code that implements them. Information hiding? Make client code less dependent on an entity's implementation details. An appropriate concern for dependency management is reflected in avoiding shared state (Item 10), applying information hiding (Item 11), and much more.

Our vote for the most valuable Item in this section goes to Item 6: Correctness, simplicity, and clarity come first. That they really, really must.

5. Give one entity one cohesive responsibility.

Summary

Focus on one thing at a time: Prefer to give each entity (variable, class, function, namespace, module, library) one well-defined responsibility. As an entity grows, its scope of responsibility naturally increases, but its responsibility should not diverge.

Discussion

A good business idea, they say, can be explained in one sentence. Similarly, each program entity should have one clear purpose.

An entity with several disparate purposes is generally disproportionately harder to use, because it carries more than the sum of the intellectual overhead, complexity, and bugs of its parts. Such an entity is larger (often without good reason) and harder to use and reuse. Also, such an entity often offers crippled interfaces for any of its specific purposes because the partial overlap among various areas of functionality blurs the vision needed for crisply implementing each.

Entities with disparate responsibilities are typically hard to design and implement. "Multiple responsibilities" frequently implies "multiple personalities"—a combinatorial number of possible behaviors and states. Prefer brief single-purpose functions (see also Item 39), small single-purpose classes, and cohesive modules with clean boundaries.

Prefer to build higher-level abstractions from smaller lower-level abstractions. Avoid collecting several low-level abstractions into a larger low-level conglomerate. Implementing a complex behavior out of several simple ones is easier than the reverse.

Examples

*Example 1: **realloc.*** In Standard C, **realloc** is an infamous example of bad design. It has to do too many things: allocate memory if passed **NULL,** free it if passed a zero size, reallocate it in place if it can, or move memory around if it cannot. It is not easily extensible. It is widely viewed as a shortsighted design failure.

*Example 2: **basic_string.*** In Standard C++, **std::basic_string** is an equally infamous example of monolithic class design. Too many "nice-to-have" features were added to a bloated class that tries to be a container but isn't quite, is undecided on iteration vs. indexing, and gratuitously duplicates many standard algorithms while leaving little space for extensibility. (See Item 44's Example.)

References

[HenneyO2a] • [HenneyO2b] • [McConnell93] §10.5 • [StroustrupOO] §3.8, §4.9.4, §23.4.3.1 • [SutterOO] §10, §12, §19, §23 • [SutterO2] §1 • [SutterO4] §37-40

6. Correctness, simplicity, and clarity come first.

Summary

KISS (Keep It Simple Software): Correct is better than fast. Simple is better than complex. Clear is better than cute. Safe is better than insecure (see Items 83 and 99).

Discussion

It's hard to overstate the value of simple designs and clear code. Your code's maintainer will thank you for making it understandable—and often that will be your future self, trying to remember what you were thinking six months ago. Hence such classic wisdom as:

> *Programs must be written for people to read, and only incidentally for machines to execute.* —Harold Abelson and Gerald Jay Sussman
>
> *Write programs for people first, computers second.* —Steve McConnell
>
> *The cheapest, fastest and most reliable components of a computer system are those that aren't there.* —Gordon Bell
>
> *Those missing components are also the most accurate (they never make mistakes), the most secure (they can't be broken into), and the easiest to design, document, test and maintain. The importance of a simple design can't be overemphasized.* —Jon Bentley

Many of the Items in this book naturally lead to designs and code that are easy to change, and clarity is the most desirable quality of easy-to-maintain, easy-to-refactor programs. What you can't comprehend, you can't change with confidence.

Probably the most common tension in this area is between code clarity and code optimization (see Items 7, 8, and 9). When—not if—you face the temptation to optimize prematurely for performance and thereby pessimize clarity, recall Item 8's point: It is far, far easier to make a correct program fast than it is to make a fast program correct.

Avoid the language's "dusty corners." Use the simplest techniques that are effective.

Examples

Example 1: Avoid gratuitous/clever operator overloading. One needlessly weird GUI library had users write w + c; to add a child control c to a widget w. (See Item 26.)

Example 2: Prefer using named variables, not temporaries, as constructor parameters. This avoids possible declaration ambiguities. It also often makes the purpose of your code clearer and thus is easier to maintain. It's also often safer (see Items 13 and 31).

References

[Abelson96] • [BentleyOO] §4 • [Cargill92] pp. 91-93 • [Cline99] §3.05-06 • [Constantine95] §29 • [Keffer95] p. 17 • [Lakos96] §9.1, §10.2.4 • [McConnell93] • [MeyersOl] §47 • [StroustrupOO] §1.7, §2.1, §6.2.3, §23.4.2, §23.4.3.2 • [SutterOO] §40-41, §46 • [SutterO4] §29

7. Know when and how to code for scalability.

Summary

Beware of explosive data growth: Without optimizing prematurely, keep an eye on asymptotic complexity. Algorithms that work on user data should take a predictable, and preferably no worse than linear, time with the amount of data processed. When optimization is provably necessary and important, and especially if it's because data volumes are growing, focus on improving big-Oh complexity rather than on micro-optimizations like saving that one extra addition.

Discussion

This Item illustrates one significant balance point between Items 8 and 9, "don't optimize prematurely" and "don't pessimize prematurely." That makes this a tough Item to write, lest it be misconstrued as "premature optimization." It is not that.

Here's the background and motivation: Memory and disk capacity continue to grow exponentially; for example, from 1988 to 2004 disk capacity grew by about 112% per year (nearly 1,900-fold growth per decade), whereas even Moore's Law is just 59% per year (100-fold per decade). One clear consequence is that whatever your code does today it may be asked to do tomorrow against more data—*much* more data. A bad (worse than linear) asymptotic behavior of an algorithm will sooner or later bring the most powerful system to its knees: Just throw enough data at it.

Defending against that likely future means we want to avoid "designing in" what will become performance pits in the face of larger files, larger databases, more pixels, more windows, more processes, more bits sent over the wire. One of the big success factors in future-proofing of the C++ standard library has been its performance complexity guarantees for the STL container operations and algorithms.

Here's the balance: It would clearly be wrong to optimize prematurely by using a less clear algorithm in anticipation of large data volumes that may never materialize. But it would equally clearly be wrong to pessimize prematurely by turning a blind eye to algorithmic complexity, a.k.a. "big-Oh" complexity, namely the cost of the computation as a function of the number of elements of data being worked on.

There are two parts to this advice. First, even before knowing whether data volumes will be large enough to be an issue for a particular computation, by default avoid using algorithms that work on user data (which could grow) but that don't scale well with data unless there is a clear clarity and readability benefit to using a less scalable algorithm (see Item 6). All too often we get surprised: We write ten pieces of code thinking they'll never have to operate on huge data sets, and then we'll turn out to be perfectly right nine of the ten times. The tenth time, we'll fall into a performance pit—we know it has happened to us, and we know it has happened or will happen to you. Sure, we go fix it and ship the fix to the customer, but it would be better to avoid such embarrassment and rework. So, all things being equal (including clarity and readability), do the following up front:

- *Use flexible, dynamically-allocated data and instead of fixed-size arrays:* Arrays "larger than the largest I'll ever need" are a terrible correctness and security fallacy. (See Item *77.)* Arrays are acceptable when sizes really are fixed at compile time.
- *Know your algorithm's actual complexity:* Beware subtle traps like linear-seeming algorithms that actually call other linear operations, making the algorithm actu ally quadratic. (See Item 81 for an example.)
- *Prefer to use linear algorithms or faster wherever possible:* Constant-time complexity, such as **push_back** and hash table lookup, is perfect (see Items 76 and 80). O(log N) logarithmic complexity, such as **set/map** operations and **lower_bound** and **upper_bound** with random-access iterators, is good (see Items 76, 85, and 86). O(N) linear complexity, such as **vector::insert** and **for each,** is acceptable (see Items 76, 81, and 84).
- *Try to avoid worse-than-linear algorithms where reasonable:* For example, by default spend some effort on finding a replacement if you're facing a **O**(N log N) or **O**(N^2) algorithm, so that your code won't fall into a disproportionately deep per formance pit in the event that data volumes grow significantly. For example, this is a major reason why Item 81 advises to prefer range member functions (which are generally linear) over repeated calls of their single-element counter parts (which easily becomes quadratic as one linear operation invokes another linear operation; see Example 1 of Item 81).
- *Never use an exponential algorithm unless your back is against the wall and you really have no other option:* Search hard for an alternative before settling for an exponen tial algorithm, where even a modest increase in data volume means falling off a performance cliff.

Second, after measurements show that optimization is necessary and important, and especially if it's because data volumes are growing, focus on improving big-Oh complexity rather than on micro-optimizations like saving that one extra addition.

In sum: Prefer to use linear (or better) algorithms wherever possible. Avoid polynomial algorithms where reasonable. Avoid exponential algorithms with all your might.

References

*[Bentley00] §6, §8, Appendix 4 * [Cormen01] • [Kernighan99] §7 • [Knuth97a] • [Knuth97b] • [Knuth98] • [McConnell93] §5.1-4, §10.6 • [Murray93] §9.11 • lSedgewick98] • [Stroustrup00] §17.1.2*

8. Don't optimize prematurely.

Summary

Spur not a willing horse (Latin proverb): Premature optimization is as addictive as it is unproductive. The first rule of optimization is: Don't do it. The second rule of optimization (for experts only) is: Don't do it yet. Measure twice, optimize once.

Discussion

As [Stroustrup00] §6's introduction quotes so deliriously:

Premature optimization is the root of all evil. —Donald Knuth [quoting Hoare] *On the other hand, we cannot ignore efficiency.* —Jon Bentley

Hoare and Knuth are, of course and as always, completely correct (see Item 6 and this Item). So is Bentley (see Item 9).

We define premature optimization as making designs or code more complex, and so less readable, in the name of performance when the effort is not justified by a proven performance need (such as actual measurement and comparison against goals) and thus by definition adds no proven value to your program. All too often, unneeded and unmeasured optimization efforts don't even make the program any faster.

Always remember:

It is far, far easier to make a correct program fast than it is to make a fast program correct.

So, by default, don't focus on making code fast; focus first on making code as clear and readable as possible (see Item 6). Clear code is easier to write correctly, easier to understand, easier to refactor—and easier to optimize. Complications, including optimizations, can always be introduced later—and only if necessary.

There are two major reasons why premature optimizations frequently don't even make the program faster. First, we programmers are notoriously bad at estimating what code will be faster or smaller, and where the bottlenecks in our code will be. This includes the authors of this book, and it includes you. Consider: Modern computers feature an extremely complex computational model, often with several pipelined processing units working in parallel, a deep cache hierarchy, speculative execution, branch prediction... and that's just the CPU chip. On top of the hardware, compilers take their best guess at transforming your source code into machine code that exploits the hardware at its best. And on top of all that complication, it's... well, it's your guess. So if you go with nothing but guesswork, there is little chance your ill-targeted micro-optimizations will significantly improve things. So, optimization must be preceded by measurement; and measurement must be preceded by optimization goals. Until the need is proven, your focus should be on priority #1—writing code for humans. (When someone asks you to optimize, *do* demand proof.)

Second, in modern programs, increasingly many operations aren't CPU-bound anyway. They may be memory-bound, network-bound, disk-bound, waiting on a web service, or waiting on a database. At best, tuning application code in such operations only make the operations wait faster. It also means that the programmer wasted valuable time improving what didn't need improving instead of adding value by improving what did.

Of course, the day will come when you do need to optimize some code. When you do so, look first for an algorithmic optimization (see Item 7) and try to encapsulate and modularize the optimization (e.g., in a function or class; see Items 5 and 11), and clearly state in a comment the reason of the optimization and a reference to the algorithm used.

A common beginner's mistake is to write new code while obsessing—with pride!—over optimal execution at the cost of understandability. More often than not, this yields miles of spaghetti that, even if correct in the beginning, is hard to read and change. (See Item 6.)

It is not premature optimization to pass by reference (see Item 25), to prefer calling prefix ++ and -- (see Item 28), and use similar idioms that should just naturally flow out of our fingertips. These are not premature optimizations; they are simply avoiding premature pessimizations (see Item 9).

Examples

*Example: An **inline** irony.* Here is a simple demonstration of the hidden cost of a premature micro-optimization: Profilers are excellent at telling you, by function hit count, what functions you should have marked inline but didn't; profilers are terrible at telling you what functions you did mark inline but shouldn't have. Too many programmers "inline by default" in the name of optimization, nearly always trading higher coupling for at best dubious benefit. (This assumes that writing **inline** even matters on your compiler. See [SutterOO], [Sutter02], and [Sutter04].)

Exceptions

When writing libraries, it's harder to predict what operations will end up being used in performance-sensitive code. But even library authors run performance tests against a broad range of client code before committing to obfuscating optimizations.

References

[Bentley00] §6 • [Cline99] §13.01-09 • [Kernighan99] §7 • [Lakos96] §9.1.14 • [Meyers97] §33 • [Murray93] §9.9-10, §9.13 • [StroustrupOO] §6 introduction • [Sutter00] §30, §46 • [Sutter02] §12 • [Sutter04] §25

9. Don't pessimize prematurely.

Summary

Easy on yourself, easy on the code: All other things being equal, notably code complexity and readability, certain efficient design patterns and coding idioms should just flow naturally from your fingertips and are no harder to write than the pes-simized alternatives. This is not premature optimization; it is avoiding gratuitous pessimization.

Discussion

Avoiding premature optimization does not imply gratuitously hurting efficiency. By premature pessimization we mean writing such gratuitous potential inefficiencies as:

- Defining pass-by-value parameters when pass-by-reference is appropriate. (See Item 25.)
- Using postfix + + when the prefix version is just as good. (See Item 28.)
- Using assignment inside constructors instead of the initializer list. (See Item 48.)

It is not a premature optimization to reduce spurious temporary copies of objects, especially in inner loops, when doing so doesn't impact code complexity. Item 18 encourages variables that are declared as locally as possible, but includes the exception that it can be sometimes beneficial to hoist a variable out of a loop. Most of the time that won't obfuscate the code's intent at all, and it can actually help clarify what work is done inside the loop and what calculations are loop-invariant. And of course, prefer to use algorithms instead of explicit loops. (See Item 84.)

Two important ways of crafting programs that are simultaneously clear and efficient are to use abstractions (see Items 11 and 36) and libraries (see Item 84). For example, using the standard library's **vector, list, map, find, sort** and other facilities, which have been standardized and implemented by world-class experts, not only makes your code clearer and easier to understand, but it often makes it faster to boot.

Avoiding premature pessimization becomes particularly important when you are writing a library. You typically can't know all contexts in which your library will be used, so you will want to strike a balance that leans more toward efficiency and reusability in mind, while at the same time not exaggerating efficiency for the benefit of a small fraction of potential callers. Drawing the line is your task, but as Item 7 shows, the bigger fish to focus on is scalability and not a little cycle-squeezing.

References

[Keffer95] pp.12-13 • [Stroustrup00] §6 introduction • [Sutter00] §6

10. Minimize global and shared data.

Summary

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

Discussion

This statement is more general than Item 18's specific treatment.

Avoid data with external linkage at namespace scope or as static class members. These complicate program logic and cause tighter coupling between different (and, worse, distant) parts of the program. Shared data weakens unit testing because the correctness of a piece of code that uses shared data is conditioned on the history of changes to the data, and further conditions the functioning of acres of yet-unknown code that subsequently uses the data further.

Names of objects in the global namespace additionally pollute the global namespace.

If you must have global, namespace-scope, or static class objects, be sure to initialize such objects carefully. The order of initialization of such objects in different compilation units is undefined, and special techniques are needed to handle it correctly (see References). The order-of-initialization rules are subtle; prefer to avoid them, but if you do have to use them then know them well and use them with great care.

Objects that are at namespace scope, static members, or shared across threads or processes will reduce parallelism in multithreaded and multiprocessor environments and are a frequent source of performance and scalability bottlenecks. (See Item 7.) Strive for "shared-nothing;" prefer communication (e.g., message queues) over data sharing.

Prefer low coupling and minimized interactions between classes. (See [Cargill92].)

Exceptions

The program-wide facilities **cin, cout,** and **cerr** are special and are implemented specially. A factory has to maintain a registry of what function to call to create a given type, and there is typically one registry for the whole program (but preferably it should be internal to the factory rather than a shared global object; see Item 11).

Code that does share objects across threads should always serialize all access to those shared objects. (See Item 12 and [Sutter04c].)

References

[Cargill92] pp. 126.136, 169-173 • [DewhurstO3] §3 • [Lakos96] §2.3.1 • [McConnell93] §5.1-4 • [Stroustrup00] §C.10.1 • [Sutter00] §47 • [SutterO2] §16, Appendix A • [Sutter04c] • [SuttHysl03]

11. Hide information.

Summary

Don't tell: Don't expose internal information from an entity that provides an abstraction.

Discussion

To minimize dependencies between calling code that manipulates an abstraction and the abstraction's implementation(s), data that is internal to the implementation must be hidden. Otherwise, calling code can access—or, worse, manipulate—that information, and the intended-to-be-internal information has leaked into the abstraction on which calling code depends. Expose an abstraction (preferably a domain abstraction where available, but at least a get/set abstraction) instead of data.

Information hiding improves a project's cost, schedule, and/or risk in two main ways:

- *It localizes changes:* Information hiding reduces the "ripple effect" scope of changes, and therefore their cost.
- *It strengthens invariants:* It limits the code responsible for maintaining (and, if it is buggy, possibly breaking) program invariants. (See Item 41.)

Don't expose data from any entity that provides an abstraction (see also Item 10). Data is just one possible incarnation of abstract, conceptual state. If you focus on concepts and not on their representations you can offer a suggestive interface and tweak implementation at will—such as caching vs. computing on-the-fly or using various representations that optimize certain usage patterns (e.g., polar vs. Cartesian).

A common example is to never expose data members of class types by making them **public** (see Item 41) or by giving out pointers or handles to them (see Item 42), but this applies equally to larger entities such as libraries, which must likewise not expose internal information. Modules and libraries likewise prefer to provide interfaces that define abstractions and traffic in those, and thereby allow communication with calling code to be safer and less tightly coupled than is possible with data sharing.

Exceptions

Testing code often needs white-box access to the tested class or module.

Value aggregates ("C-style **structs")** that simply bundle data without providing any abstraction do not need to hide their data; the data is the interface. (See Item 41.)

References

[Brooks95] §19 [McConnell93] §6.2 • [ParnasO2] • [StroustrupOO] §24.4 • [SuttHyslO4a]*

12. Know when and how to code for concurrency.

Summary

Th$_{sfl}$reaygd/y. if your application uses multiple threads or processes, know how to minimize sharing objects where possible (see Item 10) and share the right ones safely.

Discussion

Threading is a huge domain. This Item exists because that domain is important and needs to be explicitly acknowledged, but one Item can't do it justice and we will only summarize a few essentials; see the References for many more details and techniques. Among the most important issues are to avoid deadlocks, livelocks, and malign race conditions (including corruption due to insufficient locking).

The C++ Standard says not one word about threads. Nevertheless, C++ is routinely and widely used to write solid multithreaded code. If your application shares data across threads, do so safely:

- *Consult your target platforms' documentation for local synchronization primitives:* Typical ones range from lightweight atomic integer operations to memory barri ers to in-process and cross-process mutexes.
- *Prefer to wrap the platform's primitives in your own abstractions:* This is a good idea especially if you need cross-platform portability. Alternatively, you can use a li brary (e.g., pthreads [Butenhof97]) that does it for you.
- *Ensure that the types you are using are safe to use in a multithreaded program:* In par ticular, each type must at minimum:
  - *Guarantee that unshared objects are independent:* Two threads can freely use dif ferent objects without any special action on the caller's part.
  - *Document what the caller needs to do to use the same object of that type in different threads:* Many types will require you to serialize access to such shared ob jects, but some types do not; the latter typically either design away the lock ing requirement, or they do the locking internally themselves, in which case, you still need to be aware of the limits of what the internal locking granular ity will do.

  Note that the above applies regardless of whether the type is some kind of string type, or an STL container like a **vector,** or any other type. (We note that some authors have given advice that implies the standard containers are somehow special. They are not; a container is just another object.) In particular, if you want to use standard library components (e.g., **string,** containers) in a multithreaded program, consult your standard library implementation's documentation to see whether that is supported, as described earlier.

When authoring your own type that is intended to be usable in a multithreaded program, you must do the same two things: First, you must guarantee that different threads can use different objects of that type without locking (note: a type with modifiable static data typically can't guarantee this). Second, you must document what users need to do in order to safely use the same object in different threads; the fundamental design issue is how to distribute the responsibility of correct execution (race- and deadlock-free) between the class and its client. The main options are:

- *External locking: Callers are responsible for locking.* In this option, code that uses an object is responsible for knowing whether the object is shared across threads and, if so, for serializing all uses of the object. For example, string types typically use external locking (or immutability; see the third option on the next page).
- *Internal locking: Each object serializes all access to itself, typically by locking every pub lic member function, so that callers may not need to serialize uses of the object.* For ex ample, producer/consumer queues typically use internal locking, because their whole *raison d'etre* is to be shared across threads, and their interfaces are de signed so that the appropriate level of locking is for the duration of individual member function calls **(Push, Pop).** More generally, note that this option is ap propriate only when you know two things:

  First, you must know up front that objects of the type will nearly always be shared across threads, otherwise you'll end up doing needless locking. Note that most types don't meet this condition; the vast majority of objects even in a heavily multithreaded program are never shared across threads (and this is good; see Item 10).

  Second, you must know up front that per-member-function locking is at the right granularity and will be sufficient for most callers. In particular, the type's interface should be designed in favor of coarse-grained, self-sufficient operations. If the caller typically needs to lock several operations, rather than *an* operation, this is inappropriate; individually locked functions can only be assembled into a larger-scale locked unit of work by adding more (external) locking. For example, consider a container type that returns an iterator that could become invalid before you could use it, or provides a member algorithm like **find** that can return a correct answer that could become the wrong answer before you could use it, or has users who want to write **if(c.emptyO) c.push_back(x);.** (See [SutterO2] for additional examples.) In such cases, the caller needs to perform external locking anyway in order to get a lock whose lifetime spans multiple individual member function calls, and so internal locking of each member function is needlessly wasteful.

So, internal locking is tied to the type's public interface: Internal locking becomes appropriate when the type's individual operations are complete in themselves; in other words, the type's level of abstraction is raised and expressed and encapsulated more precisely (e.g., as a producer-consumer queue rather than a plain vector). Combining primitive operations together to form coarser common operations is the approach needed to ensure meaningful but simple function calls. Where combinations of primitives can be arbitrary and you cannot capture the reasonable set of usage scenarios in one named operation, there are two alternatives: a) use a callback-based model (i.e., have the caller call a single member function, but pass in the task they want performed as a command or function object; see Items 87 to 89); or b) expose locking in the interface in some way.

- *Lock-free designs, including immutability (read-only objects): No locking needed.* It is possible to design types so that no locking at all is needed (see References). One common example is immutable objects, which do not need to be locked because they never change; for example, for an immutable string type, a string object is never modified once created, and every string operation results in the creation of a new string.

Note that calling code should not need to know about your types' implementation details (see Item 11). If your type uses under-the-covers data-sharing techniques (e.g., copy-on-write), you do not need to take responsibility for all possible thread safety issues, but you *must* take responsibility for restoring "just enough" thread safety to guarantee that calling code will be correct if it performs its usual duty of care: The type must be as safe to use as it would be if it didn't use covert implementation-sharing. (See [SutterO4c].) As noted, all properly written types must allow manipulation of distinct visible objects in different threads without synchronization.

Particularly if you are authoring a widely-used library, consider making your objects safe to use in a multithreaded program as described above, but without added overhead in a single-threaded program. For example, if you are writing a library containing a type that uses copy-on-write, and must therefore do at least some internal locking, prefer to arrange for the locking to disappear in single-threaded builds of your library (#ifdefs and no-op implementations are common strategies).

When acquiring multiple locks, avoid deadlock situations by arranging for all code that acquires the same locks to acquire them in the same order. (Releasing the locks can be done in any order.) One solution is to acquire locks in increasing order by memory address; addresses provide a handy, unique, application-wide ordering.

References

[AlexandrescuO2a] • *[AlexandrescuO4]* • *[Butenhof97]* • *[HenneyOO]* • *[HenneyOl]* • *[MeyersO4]* • *[SchmidtOl]* • *[StroustrupOO] §14.9* • *[SutterO2] §16* • *[SutterO4c]*

13. Ensure resources are owned by objects. Use explicit RAII and smart pointers.

Summary

Don't saw by hand when you have power tools: C++'s "resource acquisition is initialization" (RAII) idiom is *the* power tool for correct resource handling. RAII allows the compiler to provide strong and automated guarantees that in other languages require fragile hand-coded idioms. When allocating a raw resource, immediately pass it to an owning object. Never allocate more than one resource in a single statement.

Discussion

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.** This makes a stack-based (or reference-counted) object with a resource-acquiring constructor and a resource-releasing destructor an excellent tool for automating resource management and cleanup.

The automation is easy to implement, elegant, low-cost, and inherently error-safe. If you choose not to use it, you are choosing the nontrivial and attention-intensive task of pairing the calls correctly by hand, including in the presence of branched control flows and exceptions. Such C-style reliance on micromanaging resource deallocation is unacceptable when C++ provides direct automation via easy-to-use RAII.

Whenever you deal with a resource that needs paired acquire/release function calls, encapsulate that resource in an object that enforces pairing for you and performs the resource release in its destructor. For example, instead of calling a pair of **Open-Port/ClosePort** nonmember functions directly, consider:

```
class Port {
public:
  Port( const strings destination );    // call OpenPort
  ~ Port();                             // call ClosePort
 //... ports can't usually be cloned, so disable copying and assignment...
};

void DoSomething() { Port
  portl( "serverl:80" );

} //can't forget to close portl; it's closed automatically at the end of the scope

shared_ptr< Port> port2 =/*... */      //port2 is closed automatically when the
                                       //last shared_ptr referring to it goes away
```

You can also use libraries that implement the pattern for you (see [AlexandrescuOOc]).

When implementing RAII, be conscious of copy construction and assignment (see Item 49); the compiler-generated versions probably won't be correct. If copying doesn't make sense, explicitly disable both by making them private and not defined (see Item 53). Otherwise, have the copy constructor duplicate the resource or reference-count the number of uses, and have the assignment operator do the same and ensure that it frees its originally held resource if necessary. A classic oversight is to free the old resource before the new resource is successfully duplicated (see Item 71).

Make sure that all resources are owned by objects. Prefer to hold dynamically allocated resources via smart pointers instead of raw pointers. Also, perform every explicit resource allocation (e.g., **new)** in its own statement that immediately gives the allocated resource to a manager object (e.g., **sharedptr);** otherwise, you can leak resources because the order of evaluation of a function's parameters is undefined. (See Item 31.) For example:

```
void Fun( shared_ptr<Widget> spl, shared_ptr<Widget> sp2 );

Fun( shared_ptr<Widget>(new Widget), shared_ptr<Widget>(new Widget) );
```

Such code is unsafe. The C++ Standard gives compilers great leeway to 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. That very nicely sets things up for a leak because if one of the constructor calls throws an exception, then the *other* object's memory will never be released! (See [SutterO2] for details.)

This subtle problem has a simple solution: Follow the advice to never allocate more than one resource in a single statement, and perform every explicit resource allocation (e.g., **new)** in its own code statement that immediately gives the resource to an owning object (e.g., **shared_ptr).** For example:

```
shared_ptr spl(new Widget), sp2(new Widget);
Fun( spl, sp2 );
```

See also Item 31 for other advantages to using this style.

Exceptions

Smart pointers can be overused. Raw pointers are fine in code where the pointed-to object is visible to only a restricted quantity of code (e.g., purely internal to a class, such as a **Tree** class's internal node navigation pointers).

References

[AlexandrescuOOc] • [Cline99] §31.03-05 • [DewhurstO3] §24, §67 • [Meyers96] §9-10 • [MilewskiOl] • [StroustrupOO] §14.3-4, §25.7, §E.3, §E.6 • [SutterOO] §16 • [SutterO2] §20-21 • [Vandevoorde03] §20.1.4