C++ Frequently Questioned Answers

This is a single page version of C++ FQA Lite. C++ is a general-purpose programming language, not necessarily suitable for your special purpose. FQA stands for "frequently questioned answers". This FQA is called "lite" because it questions the answers found in C++ FAQ Lite.

The single page version does not include most "metadata" sections such as the FQA FAQ.

Table of contents

Defective C++

This page summarizes the major defects of the C++ programming language (listing all minor quirks would take eternity). To be fair, some of the items *by themselves* could be design choices, not bugs. For example, a programming language doesn't have to provide garbage collection. It's the *combination* of the things that makes them *all* problematic. For example, the lack of garbage collection makes C++ exceptions and operator overloading inherently defective. Therefore, the problems are not listed in the order of "importance" (which is subjective anyway - different people are hit the hardest by different problems). Instead, most defects are followed by one of their complementary defects, so that when a defect causes a problem, the next defect in the list makes it worse.

- Implicitly called & generated functions

No compile time encapsulation

In naturally written C++ code, changing the private members of a class requires recompilation of the code using the class. When the class is used to instantiate member objects of other classes, the rule is of course applied recursively.

This makes C++ interfaces very unstable - a change invisible at the interface level still requires to rebuild the calling code, which can be very problematic when that code is not controlled by whoever makes the change. So shipping C++ interfaces to customers can be a bad idea.

Well, at least when all relevant code is controlled by the same team of people, the only problem is the frequent rebuilds of large parts of it. This wouldn't be too bad by itself with almost any language, but C++ has...

Outstandingly complicated grammar

"Outstandingly" should be interpreted literally, because *all popular languages* have context-free (or "nearly" context-free) grammars, while C++ has undecidable grammar. If you like compilers and parsers, you probably know what this means. If you're not into this kind of thing, there's a simple example showing the problem with parsing C++: is `AA BB(CC);` an object definition or a function declaration? It turns out that the answer depends heavily on the code *before* the statement - the "context". This shows (on an intuitive level) that the C++ grammar is quite context-sensitive.

In practice, this means three things. First, C++ compiles slowly (the complexity takes time to deal with). Second, when it doesn't compile, the error messages are frequently incomprehensible (the smallest error which a human reader wouldn't notice completely confuses the compiler). And three, parsing C++ right is very hard, so different compilers will interpret it differently, and tools like debuggers and IDEs periodically get awfully confused.

And *slow compilation* interacts badly with *frequent recompilation*. The latter is caused by the lack of encapsulation mentioned above, and the problem is amplified by the fact that C++ has...

No way to locate definitions

OK, so before we can parse `AA BB(CC);`, we need to find out whether `CC` is defined as an object or a type. So let's locate the definition of `CC` and move on, right?

This would work in most modern languages, in which `CC` is either defined in the same module (so we've already compiled it), or it is imported from another module (so either we've already compiled it, too, or this must be the first time we bump into that module - so let's compile it now, *once*, but of course *not* the next time we'll need it). So to compile a program, we need to compile each module, once, no matter how many times each module is used.

In C++, things are different - there are no modules. There are *files*, each of which can contain many different definitions or just small parts of definitions, and there's *no way* to tell in which files `CC` is defined, or which files must be parsed in order to "understand" its definition. So who is responsible to arrange all those files into a sensible string of C++ code? *You*, of course! In each compiled file, you `#include` a bunch of header files (which themselves include other files); the `#include` directive basically issues a copy-and-paste operation to the C preprocessor, inherited by C++ without changes. The compiler then parses the result of all those copy-and-paste operations. So to compile a program, we need to compile each file *the number of times it is used in other files*.

This causes two problems. First, it multiplies the long time it takes to compile C++ code by the number of times it's used in a program. Second, the only way to figure out what *should* be recompiled after a change to the code is to check which of the `#include` files have been changed since the last build. The set of files to rebuild generated by this inspection is usually a superset of the files that *really* must be recompiled according to the C++ rules of dependencies between definitions. That's because most files `#include` definitions they don't really need, since people can't spend all their time removing redundant inclusions.

Some compilers support "precompiled headers" - saving the result of the parsing of "popular" header files to some binary file and quickly loading it instead of recompiling from scratch. However, this only works well with definitions that almost never change, typically third-party libraries.

And now that you've waited all that time until your code base recompiles, it's time to run and test the program, which is when the next problem kicks in.

No run time encapsulation

Programming languages have rules defining "valid" programs - for example, a valid program shouldn't divide by zero or access the 7th element of an array of length 5. A valid program isn't necessarily correct (for example, it can delete a file when all you asked was to move it). However, an invalid program is necessarily incorrect (there is no 7th element in the 5-element array). The question is, what happens when an invalid program demonstrates its invalidity by performing a meaningless operation?

If the answer is something like "an exception is raised", your program runs in a managed environment. If the answer is "anything can happen", your program runs somewhere else. In particular, C and C++ are not designed to run in managed environments (think about pointer casts), and while in theory they could run there, in practice all of them run elsewhere.

So what happens in a C++ program with the 5-element array? Most frequently, you access something at the address that *would* contain the 7th element, but since there isn't any, it contains something else, which just happens to be located there. Sometimes you can tell from the source code what that is, and sometimes you can't. Anyway, you're *really* lucky if the program crashes; because if it keeps running, you'll have hard time understanding why it ends up crashing or misbehaving *later*. If it doesn't scare you (you debugged a couple of buffer overflows and feel confident), wait until you get to many megabytes of machine code and many months of execution time. That's when the real fun starts.

Now, the ability of a piece of code to modify a random object when in fact it tries to access an unrelated array indicates that C++ has no run time encapsulation. Since it doesn't have compile time encapsulation, either, one can wonder why it calls itself object-oriented. Two possible answers are warped perspective and marketing (these aren't mutually exclusive).

But if we leave the claims about being object-oriented aside, the fact that a language runs in unmanaged environments can't really be called a "bug". That's because managed environments check things at run time to prevent illegal operations, which translates to a certain (though frequently overestimated) performance penalty. So when performance isn't that important, a managed environment is the way to go. But when it's critical, you just have to deal with the difficulties in debugging. However, C++ (compared to C, for example) makes *that* much harder that it already has to be, because there are...

No binary implementation rules

When an invalid program finally crashes (or enters an infinite loop, or goes to sleep forever), what you're left with is basically the binary snapshot of its state (a common name for it is a "core dump"). You have to make sense of it in order to find the bug. Sometimes a debugger will show you the call stack at the point of crash; frequently that information is overwritten by garbage. Other things which can help the debugger figure things out may be overwritten, too.

Now, figuring out the meaning of partially corrupted memory snapshots is definitely not the most pleasant way to spend one's time. But with unmanaged environments you *have* to do it and it *can* be done, *if* you know how your source code maps to binary objects and code. Too bad that with C++, there's a ton of these rules and each compiler uses different ones. Think about exception handling or various kinds of inheritance or virtual functions or the layout of standard library containers. In C, there's no standard binary language implementation rules, either, but it's an order of magnitude simpler and in practice compilers use the same rules. Another reason making C++ code hard to debug is the above-mentioned complicated grammar, since debuggers frequently can't deal with many language features (place breakpoints in templates, parse pointer casting commands in data display windows, etc.).

The lack of a standard ABI (application binary interface) has another consequence - it makes shipping C++ interfaces to other teams / customers impractical since the user code won't work unless it's compiled with the same tools and build options. We've already seen another source of this problem - the instability of binary interfaces due to the lack of compile time encapsulation.

The two problems - with debugging C++ code and with using C++ interfaces - don't show up until your project grows complicated in terms of code and / or human interactions, that is, until it's too late. But wait, couldn't you deal with both problems programmatically? You could generate C or other wrappers for C++ interfaces *and* write programs automatically shoveling through core dumps and deciphering the non-corrupted parts, using something called reflection. Well, actually, you couldn't, not in a reasonable amount of time - there's...

No reflection

It is impossible to programmatically iterate over the methods or the attributes or the base classes of a class in a portable way defined by the C++ standard. Likewise, it is impossible to programmatically determine the type of an object (for dynamically allocated objects, this can be justified to an extent by performance penalties of RTTI, but not for statically allocated globals, and if you could *start* at the globals, you could decipher lots of memory pointed by them). Features of this sort - when a program can access the structure of programs, in particular, its own structure - are collectively called reflection, and C++ doesn't have it.

As mentioned above, this makes generating wrappers for C++ classes and shoveling through memory snapshots a pain, but that's a small fraction of the things C++ programmers are missing due to this single issue. Wrappers can be useful not only to work around the problem of shipping C++ interfaces - you could automatically handle things like remote procedure calls, logging method invocations, etc. A very common application of reflection is serialization - converting objects to byte sequences and vice versa. With reflection, you can handle it for all types of objects with the same code - you just iterate over the attributes of compound objects, and only need special cases for the basic types. In C++, you must maintain serialization-related code and/or data structures for every class involved.

But perhaps we could deal with *this* problem programmatically then? After all, debuggers do manage to display objects somehow - the debug information, emitted in the format supported by your tool chain, describes the members of classes and their offsets from the object base pointer and all that sort of meta-data. If we're stuck with C++, perhaps we could parse this information and thus have non-standard, but working reflection? Several things make this pretty hard - not all compilers can produce debug information *and* optimize the program aggressively enough for a release build, not all debug information formats are documented, and then in C++, we have a...

Very complicated type system

In C++, we have standard and compiler-specific built-in types, structures, enumerations, unions, classes with single, multiple, virtual and non-virtual inheritance, `const` and `volatile` qualifiers, pointers, references and arrays, `typedef`s, global and member functions and function pointers, and *templates*, which can have specializations on (again) *types* (or integral constants), and you can "partially specialize" templates by *pattern matching their type structure* (for example, have a specialization for `std::vector<MyRetardedTemplate<T> >` for arbitrary values of `T`), and each template can have base classes (in particular, it can be *derived from its own instantiations recursively*, which is a *well-known practice documented in books*), and inner `typedef`s, and... We have lots of kinds of types.

Naturally, representing the types used in a C++ program, say, in debug information, is not an easy task. A trivial yet annoying manifestation of this problem is the expansion of `typedef`s done by debuggers when they show objects (and compilers when they produce error messages - another reason why these are so cryptic). You may think it's a `StringToStringMap`, but only until the tools enlighten you - it's actually more of a...

```
// don't read this, it's impossible. just count the lines
std::map<std::basic_string<char, std::char_traits<char>, std::allocator<char> >,
std::basic_string<char, std::char_traits<char>, std::allocator<char> >,
std::less<std::basic_string<char, std::char_traits<char>, std::allocator<char> >
  >, std::allocator<std::pair<std::basic_string<char, std::char_traits<char>,
std::allocator<char> > const, std::basic_string<char, std::char_traits<char>,
std::allocator<char> > > > > >
```

But wait, there's more! C++ supports a wide variety of explicit and implicit *type conversions*, so now we have a nice set of rules describing the cartesian product of all those types, specifically, how conversion should be handled for each pair of types. For example, if your function accepts `const std::vector<const char*>&` (which is supposed to mean "a reference to an immutable vector of pointers to immutable built-in strings"), and I have a `std::vector<char*>` object ("a mutable vector of mutable built-in strings"), then I can't pass it to your function because the types aren't convertible. You *have* to admit that it *doesn't make any sense*, because your function guarantees that it won't change anything, and I guarantee that I don't even mind having anything changed, and still the C++ type system gets in the way and the only sane workaround is to *copy the vector*. And this is an *extremely simple* example - no virtual inheritance, no user-defined conversion operators, etc.

But conversion rules by themselves are still not the worst problem with the complicated type system. The worst problem is the...

Very complicated type-based binding rules

Types lie at the core of the C++ *binding rules*. "Binding" means "finding the program entity corresponding to a name mentioned in the code". When the C++ compiler compiles something like `f(a,b)` (or even `a+b`), it relies on the argument types to figure out which version of `f` (or `operator+`) to call. This includes overload resolution (is it `f(int,int)` or `f(int,double)`?), the handling of function template specializations (is it `template<class T> void f(vector<T>&,int)` or `template<class T> void f(T,double)`?), and the argument-dependent lookup (ADL) in order to figure out the namespace (is it `A::f` or `B::f`?).

When the compiler "succeeds" (translates source code to object code), it doesn't mean that *you* are equally successful (that is, you think `a+b` called what the compiler thought it called). When the compiler "fails" (translates source code to error messages), most humans also fail (to understand these error messages; multiple screens listing all available overloads of things like `operator<<` are less than helpful). By the way, the C++ FAQ has very few items related to the unbelievably complicated static binding, like overload resolution or ADL or template specialization. Presumably people get too depressed to ask any questions and silently give up.

In short, the complicated type system interacts very badly with *overloading* - having multiple functions with the same name and having the compiler figure out which of them to use based on the argument types (don't confuse it with *overriding* - `virtual` functions, though very far from perfect, do follow rules quite sane by C++ standards). And probably the worst kind of overloading is...

Defective operator overloading

C++ operator overloading has all the problems of C++ function overloading (incomprehensible overload resolution rules), and then some. For example, overloaded operators have to return their results by value - naively returning references to objects allocated with `new` would cause temporary objects to "leak" when code like `a+b+c` is evaluated. That's because C++ doesn't have garbage collection, since that, folks, is inefficient. Much better to have your code copy massive temporary objects and hope to have them optimized out by our friend the clever compiler. Which, of course, won't happen any time soon.

Like several other features in C++, operator overloading is not necessarily a bad thing *by itself* - it just happens to interact really badly with other things C++. The lack of automatic memory management is one thing making operator overloading less than useful. Another such thing is...

Defective exceptions

Consider error handling in an overloaded operator or a constructor. You can't use the return value, and setting/reading error flags may be quite cumbersome. How about throwing an exception?

This could be a good idea in some cases if C++ exceptions were any good. They aren't, and can't be - as usual, because of another C++ "feature", the oh-so-efficient manual memory management. If we use exceptions, we have to write exception-safe code - code which frees all resources when the control is transferred from the point of failure (`throw`) to the point where explicit error handling is done (`catch`). And the vast majority of "resources" happens to be *memory*, which is managed manually in C++. To solve this, you are supposed to use RAII, meaning that all pointers have to be "smart" (be wrapped in classes freeing the memory in the destructor, and then you have to design their copying semantics, and...). Exception safe C++ code is almost infeasible to achieve in a non-trivial program.

Of course, C++ exceptions have other flaws, following from *still other* C++ misfeatures. For example, the above-mentioned lack of reflection in the special case of exceptions means that when you catch an exception, you can't get the call stack describing the context where it was thrown. This means that debugging illegal pointer dereferencing may be easier than figuring out why an exception was thrown, since a debugger *will* list the call stack in many cases of the former.

At the bottom line, `throw/catch` are about as useful as `longjmp/setjmp` (BTW, the former typically runs faster, but it's mere *existence* makes *the rest of the code* run slower, which is almost never acknowledged by C++ aficionados). So we have two features, each with its own flaws, and no interoperability between them. This is true for the vast majority of C++ features - most are...

Duplicate facilities

If you need an array in C++, you can use a C-like `T arr[]` or a C++ `std::vector<T>` or any of the array classes written before `std::vector` appeared in the C++ standard. If you need a string, use `char*` or `std::string` or any of the pre-standard string classes. If you need to take the address of an object, you can use a C-like pointer, `T*`, or a C++ reference, `T&`. If you need to initialize an object, use C-like aggregate initialization or C++ constructors. If you need to print something, you can use a C-like `printf` call or a C++ `iostream` call. If you need to generate many similar definitions with some parameters specifying the differences between them, you can use C-like macros or C++ templates. And so on.

Of course you can do the same thing in many ways in almost any language. But the C++ feature duplication is quite special. First, the many ways to do the same thing are usually not purely syntactic options directly supported by the compiler - you can compute `a+b` with `a-b*-1`, but that's different from having `T*` and `T&` in the same language. Second, you probably noticed a pattern - C++ adds features duplicating functionality already in C. This is bad by itself, because the features don't interoperate well (you can't `printf` to an `iostream` and vice versa, code mixing `std::string` and `char*` is littered with casts and calls to `std::string::c_str`, etc.). This is made even worse by the *pretty amazing* fact that the new C++ features are actually *inferior* to the old C ones in many aspects.

And the best part is that C++ devotees *dare* to refer to the C features as evil, and frequently will actually resort to finger pointing and name calling when someone uses them in C++ code (not to mention using plain C)! And *at the same time* they (falsely) claim that C++ is compatible with C and it's one of its strengths (why, if C is so evil?). The real reason to leave the C syntax in C++ was of course marketing - there's absolutely NO technical reason to *parse C-like syntax* in order to *work with existing C code* since that code can be compiled separately. For example, mixing C and the D programming language isn't harder than mixing C and C++. D is a good example since its stated goals are similar to those of C++, but almost all other popular languages have ways to work with C code.

So IMO all that old syntax was kept for strictly commercial purposes - to market the language to non-technical managers or programmers who should have known better and didn't understand the difference between "syntax" and "compatibility with existing code" and simply asked whether the old code will compile with this new compiler. Or maybe they thought it would be easier to learn a pile of new syntax when you also have the (smaller) pile of old syntax than when you have just the new syntax. Either way, C++ got wide-spread by exploiting misconceptions.

Well, it doesn't matter anymore why they kept the old stuff. What matters is that the new stuff isn't really new, either - it's obsessively built in ways exposing the C infrastructure underneath it. And *that* is purely a wrong design decision, made without an axe to grind. For example, in C++ there's...

No high-level built-in types

C is a pretty low-level language. Its atomic types are supposed to fit into machine registers (usually one, sometimes two of them). The compound types are designed to occupy a flat chunk of memory with of a size known at compile time.

This design has its virtues. It makes it relatively easy to estimate the performance & resource consumption of code. And when you have hard-to-catch low-level bugs, which sooner or later happens in unmanaged environments, having a relatively simple correspondence between source code definitions and machine memory helps to debug the problem. However, in a high-level language, which is supposed to be used when the development-time-cost / execution-time-cost ratio is high, you need things like resizable arrays, key-value mappings, integers that don't overflow and other such gadgets. Emulating these in a low-level language is possible, but is invariably painful since the tools don't understand the core types of your program.

C++ doesn't add any built-in types to C *(correction)*. All higher-level types must be implemented as user-defined classes and templates, and this is when the defects of C++ classes and templates manifest themselves in their full glory. The lack of syntactic support for higher-level types (you can't initialize `std::vector` with `{1,2,3}` or initialize an `std::map` with something like `{"a":1,"b":2}` or have large integer constants like `3453485348545459347376`) is the small part of the problem. Cryptic multi-line or *multi-screen* compiler error messages, debuggers that can't display the standard C++ types and slow build times unheard of anywhere outside of the C++ world are the larger part of the problem. For example, here's a simple piece of code using the C++ standard library followed by an error message produced from it by gcc 4.2.0. Quiz: what's the problem?

```
// the code
typedef std::map<std::string,std::string> StringToStringMap;
void print(const StringToStringMap& dict) {
  for(StringToStringMap::iterator p=dict.begin(); p!=dict.end(); ++p) {
    std::cout << p->first << " -> " << p->second << std::endl;
  }
}
// the error message
test.cpp: In function 'void print(const StringToStringMap&)':
test.cpp:8: error: conversion from
'std::_Rb_tree_const_iterator<std::pair<const std::basic_string<char,
std::char_traits<char>, std::allocator<char> >, std::basic_string<char,
std::char_traits<char>, std::allocator<char> > > >' to non-scalar type
'std::_Rb_tree_iterator<std::pair<const std::basic_string<char,
std::char_traits<char>, std::allocator<char> >, std::basic_string<char,
std::char_traits<char>, std::allocator<char> > > >' requested
```

The decision to avoid new built-in types yields other problems, such as the ability to throw anything, but without the ability to *catch* it later. `class Exception`, a built-in base class for all exception classes treated specially by the compiler, could solve this problem with C++ exceptions (but not others). However, the most costly problem with having no new high-level built-in types is probably the lack of easy-to-use containers. But to have those, we need more than just new built-in types and syntax in the C++ compiler. Complicated data structures can't be manipulated easily when you only have...

Manual memory management

Similarly to low-level built-in types, C++ manual memory management is inherited from C without changes (but with the mandatory addition of duplicate syntax - `new/delete`, which normally call `malloc/free` but don't have to do that, and of course can be overloaded).

Similarly to the case with low-level built-in types, what makes sense for a low-level language doesn't work when you add higher-level features. Manual memory management is incompatible with features such as exceptions & operator overloading, and makes working with non-trivial data structures very hard, since you have to worry about the life cycles of objects so they won't leak or die while someone still needs them.

The most common solution is copying - since it's dangerous to point to an object which can die before we're done with it, make yourself a copy and become an "owner" of that copy to control its life cycle. An "owner" is a C++ concept not represented in its syntax; an "owner" is the object that is responsible to deallocate a dynamically allocated chunk of memory or some other resource. The standard practice in C++ is to assign each "resource" (a fancy name for memory, most of the time) to an owner object, which is supposed to prevent resource leaks. What it doesn't prevent is access to dead objects; we have copying for that. Which is slow and doesn't work when you need many pointers to *the same* object (for example, when you want other modules to see your modifications to the object).

An alternative solution to copying is using "smart" pointer classes, which could emulate automatic memory management by maintaining reference counts or what-not. To implement the pointer classes for the many different types in your program, you're encouraged to use...

Defective metaprogramming facilities

There are roughly two kinds of metaprogramming: code that generates other code and code that processes other code. The second kind is practically impossible to do with C++ code - you can't reliably process source code due to the extremely complicated grammar and you can't portably process compiled code because there's no reflection. So this section is about the first kind - code generation.

You can generate C++ code from within a C++ program using C macros and C++ templates. If you use macros, you risk getting clubbed to death by C++ fanatics. Their irrational behavior left aside, these people do have a point - C macros are pretty lame. Too bad templates are probably even worse. They are limited in ways macros aren't (however, the opposite is also true). They compile forever. Being the only way to do metaprogramming, they are routinely abused to do things they weren't designed for. And they are a rats' nest of bizarre syntactic problems.

That wouldn't necessarily be so bad if C++ didn't *rely* on metaprogramming for doing essential programming tasks. One reason C++ has to do so is that in C++, the common practice is to use static binding (overload resolution, etc.) to implement polymorphism, not dynamic binding. So you can't take an arbitrary object at run time and print it, but in many programs you *can* take an arbitrary *type* at compile time and print objects of this type. Here's one common (and broken) application of metaprogramming - the ultimate purpose is to be able to print arbitrary object *at run time*:

```
// an abstract base class wrapping objects of arbitrary types.
// there can be several such classes in one large project
struct Obj {
  virtual void print(std::ostream&) const = 0;
};
template<class T> struct ObjImpl : Obj {
  T wrapped;
  virtual void print(std::ostream& out) const { out << wrapped; }
};
// now we can wrap int objects with ObjImpl<int> and string objects
// with ObjImpl<std::string>, store them in the same collection of Obj*
// and print the entire collection using dynamic polymorphism:
void print_them(const std::vector<Obj*>& objects) {
  for(int i=0; i<(int)objects.size(); ++i) {
    objects[i]->print(std::cout); // prints wrapped ints, strings, etc.
    std::cout << std::endl;
  }
}
```

Typically there are 10 more layers of syntax involved, but you get the idea. This sort of code doesn't really work because it requires all relevant overloads of `operator<<` to be visible *before* the point where `ObjImpl` is defined, and that doesn't happen unless you routinely sort your `#include` directives according to that rule. Some compilers will compile the code correctly with the rule violated, some will complain, some will silently generate wrong code.

But the most basic reason to rely on the poor C++ metaprogramming features for everyday tasks is the above-mentioned ideological decision to avoid adding high-level built-in types. For example, templates are at the core of the...

Unhelpful standard library

Most things defined by the C++ standard library are templates, and relatively sophisticated ones, causing the users to deal with quite sophisticated manifestations of the problems with templates, discussed above. In particular, a special program called STLFilt exists for *decrypting the error messages* related to the C++ standard library. Too bad it doesn't patch the debug information in a similar way.

Another problem with the standard library is all the functionality that's not there. A large part of the library duplicates the functionality from the C standard library (which is itself available to C++ programs, too). The main new thing is containers ("algorithms" like `max` and `adjacent_difference` don't count as "functionality" in my book). The standard library doesn't support listing directories, opening GUI windows or network sockets. You may think that's because these things are non-portable. Well, the standard library doesn't have matrices or regular expressions, either.

And when you use the standard library in your code, one reason it compiles slowly to a large binary image is that the library extensively uses the...

Defective inlining

First, let's define the terms.

"Inlining" in the context of compilers refers to a technique for *implementing* function calls (instead of generating a sequence calling the implementation of the function, the compiler integrates that implementation at the point where the call is made). "Inlining" in the context of C++ refers to a way to *define* functions in order to *enable* (as opposed to "force") such implementation of the calls to the function (the decision whether to actually use the opportunity is made by the compiler).

Now, the major problem with this C++ way to enable inlining is that you have to place the definition of the function in header files, and have it recompiled over and over again from source. This doesn't have to be that way - the recompilation from source can be avoided by having higher-level object file formats (the way it's done in LLVM and gcc starting from version 4). This approach - link-time inlining - is one aspect of "whole program optimization" supported by modern compilers. But the recompilation from source could also be avoided in simpler ways if C++ had a way to locate definitions instead of recompiling them, which, as we've seen, it hasn't.

The crude support for inlining, designed with a traditional implementation of a C tool chain in mind, wouldn't be as bad if it wasn't *used all the time*. People define large functions inline for two reasons. Some of them "care" (emotionally) about performance, but never actually measure it, and someone told them that inlining speeds things up, and forgot to tell them how it can slow them down. Another reason is that it's simply *annoying* to define functions non-inline, since that way, you place the full function definition in a `.cpp` file and its prototype in a `.h` file. So you write the prototype twice, *with small changes* (for example, if a class method returns an object of a type itself defined in the class, you'll need an extra namespace qualification in the `.cpp` file since you're now *outside of the namespace of the class*). Much easier to just have the body written right in the `.h` file, making the code compile more slowly and recompile more frequently (changing the function body will trigger a recompilation).

And you don't even need to actually *write* any inline functions to get most of their benefits! A large subset of the inline functions of a program are...

Implicitly called & generated functions

Here's a common "design pattern" in C++ code. You have a huge class. Sometimes there's a single pseudo-global object of this class. In that case, you get all the *drawbacks* of global variables because everybody has a pointer to the thing and modifies it and expects others to see the changes. But you get no *benefits* of global variables since the thing is allocated on the stack and when your program crashes with a buffer overflow, you can't find the object in a debugger. And at other times there are many of these objects, typically kept in a pseudo-global collection.

Anyway, this huge class has no constructors, no destructor and no `operator=`. Of course people create and destroy the objects, and sometimes even assign to them. How is this handled by the compiler?

This is handled by the compiler by generating a *gigantic* pile of code at the point where it would call the user-defined functions with magic names (such as `operator=`) if there were any. When you *crash* somewhere at that point, you get to see *kilobytes* of assembly code in the debugger, all generated from *the same source code line*. You can then try and figure out which variable didn't like being assigned to, by guessing where the class member offsets are in the assembly listing and looking for symbolic names of the members corresponding to them. Or you can try and guess who forgot all about the fact that these objects were assigned to using the "default" `operator=` and added something like built-in pointer members to the class. Because that wouldn't work, and could have caused the problem.

Implicit generation of functions is problematic because it slows compilation down, inflates the program binaries and gets in the way when you debug. But the problem with *implicitly calling* functions (whether or not they were *implicitly generated*) is arguably even worse.

When you see code like `a=f(b,c)` (or even `a=b+c`, thanks to operator overloading), you don't know whether the objects are passed by reference or by value (see "information hiding"). In the latter case, the objects are copied with implicitly called functions; in the former case, that's possible, too, if implicit type conversions were involved. Which means that you don't really understand what the program does unless you know the relevant information about the relevant overloads and types. And by the way, the fact that you can't see whether the object is passed by reference or by value at the point of call is *another* example of implicit stuff happening in C++.

One more problem with automatically generated functions (such as constructors and destructors) is that they must be *regenerated* when you add private members to a class, so changing the private parts of a class triggers recompilation... Which brings us back to square 1.

Big Picture Issues

This part deals with the Big (and somewhat Sad) Picture.

[6.1] Is C++ a practical language?

FAQ: Sure - not perfect, but mature and well-supported, which is good for business.

FQA: C++ is not "mature" in the sense that different compilers will interpret it differently, and C++ modules built by different vendors will not work with each other. C++ is not "well-supported" in the sense that development tools for C++ lack features and are unreliable compared to other languages. These things make one ask "Am I the first one trying to do this?" all the time.

This situation is not likely to change, because it follows from the C++ definition. C++ is very complicated for programs (or people) to understand. C++ specification leaves out most aspects crucial for interoperability, such as modules and calling conventions. C++ has a huge installed base, and since solving these problems backward-compatibly is impossible, they won't be solved.

[6.2] Is C++ a perfect language?

FAQ: No, and it shouldn't be, it should be practical, which, as we've just seen, it is. Perfect is for academy, practical is for business.

FQA: No language is "perfect" because our requirements from a "perfect" language are inconsistent with each other. So instead of perfection, good languages provide *consistency* and *usability*. This can be called "practical" from the point of view of language users.

C++ is different - it's designed for perfection. Where other languages give you a feature, C++ gives you meta-features. Instead of built-in strings and vectors, it gives you templates. Instead of garbage collection, it gives you smart pointers. This way, you can (theoretically) implement your own "perfect" (most efficient and generic) strings. In practice, this turns into a nightmare since many different kinds of strings, smart pointers, etc., each perfect in its own way, will not work with each other. C++ sacrifices usability for perfection.

However, despite the obsession with perfection, C++ *is* "practical" - from a language designer's perspective rather than from a user's point of view. The "practical" thing in C++ is that it's based on C. This helped the language gain popularity. This is also the main reason for inconsistencies in the language - ambiguities in the grammar (declaration/definition, type name/object name...), duplications of functionality in the different features (pointers/references, constructors/aggregate initialization, macros/constants/templates, files/namespaces...). C++ sacrifices consistency for popularity. This "practical" approach helps to increase the number of C++ users, but it doesn't help those users to get their job done.

[6.3] What's the big deal with OO?

FAQ: Object-oriented programming is the best known way to develop complex systems. It was invented because customers kept demanding increasingly complex systems.

FQA: Object-oriented programming is very useful. For a lot of things it's so useful you're likely to want support for it built into your language. Of course nobody knows how to build complex systems in the general case (or in your special case). OOP can help, other things can help, but ultimately there is no simple way to deal with complexity. Which is only surprising if you think that there should exist a reliable process to produce anything people are willing to pay for. The laws of business are powerful, the laws of nature are more powerful.

Most kinds of built-in language support for object-oriented programming, including no such support, have big advantages over C++ classes. The single biggest problem with C++ classes is that private members are written in header files, so changing them requires recompiling the code using them - for important practical purposes, this makes private members a part of the interface. C++ is built such that recompilation is very slow (an order of magnitude slower than it is with virtually any other language), and classes are built to make recompilation a frequent event.

From a business perspective, this means two things: your C++ developers spend a significant amount of their time in recompilation cycles, and C++ interfaces provided to your customers or by your vendors will cause you major headaches (when versions are upgraded, some of the code won't be recompiled and software will fail in creative ways). Luckily, C++ interfaces are *hard* to provide (effectively all parties must use the same compiler with the same settings), so quite typically C++ modules have interfaces written in C.

[6.4] What's the big deal with generic programming?

FAQ: Generic programming allows to create components which are easy to use, widely applicable (reusable) *and* efficient. Using them makes your code faster and reduces the amount of errors. Creating them is a "non-process" (a poetic description of solving hard problems follows - waking up at night and other things probably questionable from the "business perspective" of which the FAQ is so fond). Most people can use them, but aren't *cut out* to create their own - one must *like to solve puzzles* for that. But these generic components are so generic that you can probably find an off-the-shelf one for your needs.

FQA: "Generic programming" in the context of C++ refers to templates.

Templates are hard to *use* (and *not* only define & implement) due to cryptic compiler error messages, extremely long compilation time and remarkable hostility to symbolic debugging - both code browsing and data inspection. The usability problems are *not* solved by using off-the-shelf components.

Templates are mostly applicable to containers or smart pointers, which can contain or point to almost anything. When the constraints on the input are less trivial, most of the time you either don't really need polymorphism, or you are better off with dynamic polymorphism (for example, the kind you get with C++ virtual functions). That's because in most cases, the benefits (such as separate compilation) are worth the overhead of dynamic binding (which is dwarfed by the complexity of the dispatched operations themselves).

Templates are a form of code generation, and hence they don't make code faster or slower compared to code you'd write manually. They do tend to make it larger since the compiler generates the same code many times. Although there are theoretical ways to avoid this, you find yourself solving someone else's problem. With the "evil" C macros you can at least control when they are expanded.

People who like to solve puzzles usually prefer *interesting* puzzles. With templates, the greatest puzzle is what on Earth the code means (even compilers frequently disagree). Practical people avoid fiddling with problems which nobody actually wants solved, and templates are only interesting inside the world of C++, not the real world.

[6.5] Is C++ better than Ada? (or Visual Basic, C, FORTRAN, Pascal, Smalltalk, or any other language?)

FAQ: Answering this question is not very helpful because business considerations dominate technical considerations. Specifically, availability (of compile time and run time environments, tools, developers) is the most important consideration. People who don't get this are techie weenies endangering their employer's interests.

FQA: Answering this question is not very helpful because the real question is what language is best for your specific purposes. The purposes are defined by the business considerations (what seems worth doing) and by technical considerations (what seems possible to do). In particular, your purposes may limit the availability of developers, tools, etc. These constraints are necessary to meet.

One thing is always true: where you can use C++, you can use C. In particular, if someone gave you C++ interfaces, a thin layer of wrappers will hide them. Using C instead of C++ has several practical benefits: faster development cycle, reduced complexity, better support by tools such as debuggers, higher portability and interoperability. When C++ is an option, C is probably a better option.

Another thing is always true: where you can use a managed environment (where the behavior of wrong programs is defined), using it will save a lot of trouble. C++ (like C) is designed for unmanaged environments (where the behavior of wrong programs is undefined). Unmanaged environments make it very hard to locate faults and impose no limit on the damage done by an undetected fault. In theory, C++ implementations can run in managed environments, but in practice they don't because of innumerable compatibility issues.

Yet another thing is almost always true: picking up a new language is easier for an experienced C++ programmer than working in C++. This is the result of the exceeding complexity of C++.

People who think there's no point in comparing programming languages, for example because "business considerations dominate technical considerations", are free to start their new projects in COBOL (**CO**mmon **B**usiness-**O**riented **L**anguage).

[6.6] Who uses C++?

FAQ: Lots and lots and lots of people and organizations. Which is excellent for business since a lot of developers are available.

FQA: Empirical studies indicate that 20% of the people drink 80% of the beer. With C++ developers, the rule is that 80% of the developers understand at most 20% of the language. It is not *the same* 20% for different people, so don't count on them to understand each other's code.

Two things are at fault: the exceptional complexity of C++ and its wide popularity, driving hordes of people who don't consider professional competence a personal priority. The few competent developers will spend much of their time dealing with problems created by the language instead of the original problems (and a subset of these developers will not even notice).

The large number of developers at least has the advantage of motivating the development of tools for dealing with C++ code. However, the design of the language makes it notoriously hard to produce such tools - a problem motivation can't quite remedy. Compare the quality of code browsing in C++ IDEs to IDEs of other languages and you'll get the idea. You can look at language-specific IDEs, general-purpose programming IDEs or extensions for general-purpose text editors - C++ loses everywhere. Don't just look at small examples, try it on large programs (especially ones using cutting-edge template libraries).

[6.7] How long does it take to learn OO/C++?

FAQ: In 6-12 months you can become proficient, in 3 years you are a local mentor. Some people won't make it - those can't learn, and/or they are lazy. Changing the way you think and what you consider "good" is hard.

FQA: In 6-12 months you can become as proficient as it gets. It is impossible to "know" C++ - it keeps surprising one forever. For example, what does the code `cout << p` do when p is a `volatile` pointer? Hint: as experienced people might *expect*, there's an unexpected implicit type conversion involved.

While some people are better at learning than others, it is also true that some languages are easier to learn and use than others. C++ is one of the hardest, and your reward for the extra effort spent learning it is likely to be extra effort spent using it. If you find it hard to work in C++, trying another language may be a good idea.

Before you subvert the way you think about programming and your definition of "good" in this context to fit C++, it might be beneficial to ask the common sense again. For example, does compilation time *really* cost nothing (is development time that cheap, are there compilation servers with 100 GHz CPUs around)? Is run time *really* priceless (don't user keystrokes limit out speed, how much data are we processing anyway)? How efficient a C++ construct *really* is in your implementation (templates, exceptions, endless copying & conversion)? The reasoning behind C++ may be consistent, but the assumptions almost never hold.

Learning OO has nothing to do with learning C++, and it is probably better to learn OO using a different language as an example. The OO support in C++ is almost a parody on OO concepts. For example, encapsulation is supposed to hide the implementation details from the user of a class. In C++, the implementation is hidden neither at compile time (change a private member and you must recompile the calling code) nor at run time (overwrite memory where an object is stored and you'll find out a lot about the implementation of its class - although in an unpleasant way).

[6.8] What are some features of C++ from a business perspective?

FAQ: Here are a few:

- A huge installed base, which means good support

- Allows to provide simplified interfaces, reducing the defect rate
- Operator overloading reduces learning curves by exploiting intuition
- Reduces safety-vs-usability and safety-vs-speed trade-offs
- Makes it possible for old code to call new code

FQA: Here are a few more:

- No practical implementation of C++ runs in managed environments, increasing both the defect rate and the potential damage of an undetected defect
- Providing C++ interfaces to a software component is impossible in practice due to lack of compile time and run time interoperability
- C++ is extremely inconsistent and complicated, increasing learning curves and the defect rate
- C++ compilers typically fail to comply to its intricate standard, reducing portability
- C++ compilation is both very slow and very frequent, increasing development time and defect rate (people write cryptic and dangerous code to avoid recompilation, for example, use global variables instead of adding arguments to functions, saving 1.5 hours per rebuild x 20 developers = 30 hours of downtime)
- C++ lacks standard types representing basic data structures like strings, arrays and lists (or has more than one standard and many non-standard ones, which is the same), making it harder to reuse code (each interface works with a different kind of strings) and reducing the speed due to run time type conversion

All things mentioned in the FAQ are false for most practical purposes:

- Despite the "huge" installed base, the tools dealing with C++ code are poor and their interoperability is a disaster (in both cases the problem is in the language definition)
- C++ interfaces are usually very complicated (lots of small classes, implicitly generated functions like constructors & destructors, code bundled with the interface in template definitions...). As mentioned above, providing C++ interfaces to someone outside of your team is very hard in practice. And private members are for many purposes effectively a part of your interface.
- Operator overloading is almost always counter-intuitive if one tries to understand the functionality (why does the left shift operator *print* things?), and always counter-intuitive if one tries to estimate performance (go figure if `*` multiplies two integers or two matrices, especially inside a template definition) or locate bugs (lethal ones can hide in places like `operator=`, where they are hard to see)
- C++ doesn't reduce safety-vs-anything trade-off since it's extremely unsafe (it "supports" all the undefined behavior of C like buffer overflows, adds many new scenarios with undefined result like invisibility of template specializations at the point of usage, and its complexity reduces the chances that someone actually knows what a program does and can prove its correctness). Where's the "trade-off"?
- Old code can call new code in almost any popular language, for example, C (the ancient `qsort` function is probably calling new code as we speak). The item is really supposed to describe the benefits of OO to non-technical people. C++ is not likely to give its user the benefits of OO.

[6.9] Are virtual functions (dynamic binding) central to OO/C++?

FAQ: Sure, that's what makes C++ an object-oriented language. Don't switch from C to C++ unless you need virtual functions!

FQA: They probably are if you consider C++ an "object-oriented" language (a C++ debugger doesn't - try asking it to show what "object" is located at a random place, for example). You have to carefully define "object-oriented" so that C++ fits the definition.

Dynamic binding is central to any language since otherwise old code can't call new code, making code reuse very hard. Virtual functions are one form of dynamic binding supported by C++ (function pointers, inherited from C, are another one).

Switching from any language to C++ is not necessarily a good idea.

[6.10] I'm from Missouri. Can you give me a simple reason why `virtual` functions (dynamic binding) make a big difference?

FAQ: Before OO, you could only reuse old code by having new code call it. With OO, old code can call new code - more reuse. Even if the source code for the old code is not available.

FQA: It is unclear why the FAQ gets this wrong - most of the time it is technically accurate. Dynamic binding - old code calling new code - exists outside of OO. There are countless examples on any scale, ranging from the C `qsort` function to operating systems, which run programs written long after the code of those systems.

The special thing in OO is that, well, it works with objects. In the case of dynamic binding, this means that not only does old code call new code - it also passes the *state* (encapsulated in the object receiving the method call) needed for this new code to work. This also happens outside of OO, but OO is an excellent unifying framework for dealing with this kind of thing. Especially if you have a *good* OO environment.

The omission of facts in the FAQ is much more typical than the technical inaccuracy. Specifically, there's a difference between theory and practice when it comes to old code *not available in source form* calling new code. In practice, code generated from C++ source is not portable, limiting the scenarios where the reuse actually works. Worse, even C++ implementations running on the same hardware and operating system are rarely compatible. For actually having old code call new code, you must limit yourself to a small subset of the language (C is one good one), and/or have both the old and the new code built with the same tools under the same settings.

[6.11] Is C++ backward compatible with ANSI/ISO C?

FAQ: Almost. But a declaration of a function without parameters means different things in C and C++, and `sizeof('x')` is likely to yield a different value, and...

FQA: The pair of words "almost compatible" is *almost meaningless* - for many technical purposes, compatibility is a binary thing. "Compatible", on the other hand, can have *several* meanings.

If your question is "Can I compile C code with a C++ compiler?", the answer is "no" because of numerous differences in the way code is interpreted (some things will be reported by the compiler, some will be silently misinterpreted). However, this is not a real problem, since you can compile C code with a C compiler.

If your question is "Can I call C code from C++ code?", the answer is "yes", but it's not special to C++. You can call C code from virtually any popular language because most of today's environments are based on C, making it both easy and beneficial to support this.

If your question is "Can I call C++ code from C code?", the answer is "sometimes". It is possible if the C++ code exposes a C interface (no classes, no exceptions...), and even then there are problems like making sure C++ global constructors and destructors are invoked. Many platforms provide ways for this to work.

If your question is "Is it easier for a C programmer to learn and use C++ than another new language, possibly object-oriented?", the answer is "no". C++ is very hard to learn and use and the hardest parts are not related to the C subset, but to the new parts and the way they interact with the old parts.

If your question is "Are C++ programs likely to contain bugs similar to these littering C programs, like buffer overflows?", the answer is "yes". If you are willing to sacrifice performance to gain stability, a managed environment might suit your needs. If you want to improve the stability of your programs without sacrificing neither development time nor run time, you probably can't. In particular, the "high-level" C++ is *compatible* to the "low-level" C when it comes to damage caused by low-level errors.

[6.12] Is C++ standardized?

FAQ: Yes, an ISO standard was adopted in 1997. The FAQ mentions twice that it was "adopted by unanimous vote".

FQA: Yes, there is a document specifying what "C++" means, and lots of implementation vendors signed it. The important thing about standardization, however, is the practical implications. Let's examine them.

The C++ standard does not specify what source code is translated to. Unlike code built from Java source, compiled C++ code will usually only run on one hardware/OS configuration.

The C++ standard does not address interoperability between implementations. Unlike code built from C source, compiled C++ code will only be able to call C++ code built with the same compiler and the same settings. Different implementations implement exceptions, global initialization & destruction, virtual functions, RTTI, mangling conventions, etc. etc. differently. The C standard leaves out interoperability between implementations just like the C++ standard - but C is an order of magnitude simpler, so you won't have these problems in practice.

The C++ standard does not define a term like "module" or "library" - only "program" and "translation unit" (roughly, the latter means a preprocessed source file). If you deliver dynamically/statically linked libraries, you're on your own. Again, you will have problems with global initialization & destruction, RTTI, exceptions...

The C++ standard does not specify a machine-readable definition of the C++ grammar, and the question whether a given sequence of characters is legal C++ is undecidable. Building tools reliably processing C++ code (including *compilers*) is extremely hard.

The C++ standard has been out there for a long time. Today, different C++ compilers will interpret C++ code differently. The most frequent source of problems is *static binding* - figuring out what function calls should be generated from a given statement. Compilers implement name resolution (affected by namespaces, function & operator overloading, template signature matching & specialization, implicit type conversions, type qualifiers, inheritance...) differently. Neither the standard document nor common sense will easily tell you which compiler is "right".

You may think that compilers will eventually catch up with the standard (which most vendors are trying to do all the time, but the tools still frequently disagree on the question what "C++" means). Well, the *next* version of the standard is supposed to be adopted before 2010, giving those vendors some more work. For those compiler writers with *really* too much time on their hands, there's C++/CLI.

C++ is standardized, but it may have less practical benefits than you might be used to expect from "standards".

[6.13] Where can I get a copy of the ANSI/ISO C++ standard?

FAQ: Get ready to spend some money. A list of links follows.

FQA: Get ready to throw away some money. Seriously, what are you going to do with your copy? The document is incomprehensible.

The document may be useful if you are a language lawyer planning to sue the people responsible for the language or a particular implementation. But if you want to build working software, it's more practical to accept the fact that your implementation is not standard-compliant in many dark areas. If you find a front-end bug (for example, many times nifty, expensive tools will *crash* trying to process complicated C++ code; all compilers I used did) - that's actually *your* problem. While you are lost in the maze of C++ features, your competitor has already released a working product written without such complications.

The document is also useful if you're into meta-programming (compilers/debuggers/profilers/verifiers...) and want to write tools dealing with C++ code. The standard may help chill your passion before you throw away too much of your time.

[6.14] What are some "interview questions" I could ask that would let me know if candidates really know their stuff?

FAQ: If you are a non-technical person (manager/HR), ask a technical person to help you judge the technical competence of a candidate. If you are a technical person, the FAQ is one source of good questions, separating the truly competent people from the posers.

FQA: The good interview questions probably don't mention anything unique to C++.

Ultimately, you are looking for people with good will (some call them "cooperative"), who will do things, not just talk about them (some call them "practical"), and who will think, not just do (some call them "intelligent"). So the best questions, relevant for all candidates, are about their largest last projects. The answers give you lots of information and good answers are almost impossible to fake.

You may also need people to have some prior knowledge relevant to their work since you don't have time to have them trained and gain experience. If you are sure that's the case (despite the fact that the people you are looking for are good learners), ask specific questions. Questions about high-level software organization issues (like OO) may be useful. Questions about low-level software construction issues (like pointers) may be useful. These issues are not specific to C++.

Asking about things specific to C++ is not very useful.

First, many of these things are useless for any practical purpose and are best avoided. Whether someone knows these things is correlated quite loosely with proficiency, and there are many excellent developers out there who weren't confronted with a particular obscure C++ feature yet, or successfully forgot it. So chances are that you are going to reject a good candidate.

Second, a *good* candidate actually knowing the answer may prefer an employer asking more relevant and practical questions. So chances are that a good candidate is going to reject you.

And third, there are people who look for the most complicated way to solve a problem to show off their intelligence. These tend to stumble into the dark areas of the tools they use all the time, so they will know answers to many C++-specific questions (they won't know answers to many more, because almost nobody does). Your questions will rank these people as the best possible candidates. Later you will find out that these people are poor practitioners.

[6.15] What does the FAQ mean by "such and such is evil"?

FAQ: This means that a feature should be avoided whenever possible. The strong word is supposed to help people change their old thinking.

FQA: This means the feature satisfies the following conditions:

- It is inherited from C
- It is easy to abuse *(especially when it interacts with the new C++ features)*
- It can cause problems when abused *(especially when it interacts with the new C++ features)*
- C++ provides one or more facilities duplicating the functionality of the feature, replacing the original problems with new and much more complicated problems

For example, macros, pointers, and arrays meet this definition (the corresponding C++ "solutions" are const & template, references & smart pointers, and vector/string classes). Include files almost meet this definition, except that C++ doesn't duplicate this functionality (namespaces are a parallel notion of "modules", but they can't be used to locate definitions). Consequently, the FAQ will not call include files "evil". On the other hand, function overloading doesn't come from C, so duplicate facilities like template specialization, default arguments, etc. are not enough for the FAQ to call function overloading "evil". Still, function overloading is very commonly abused leading to major problems.

A C++ user is likely to have a different definition of "evil". A user doesn't care whether something came from C or not, and whether C++ tried to offer duplicate facilities (while forcing users to deal with the original ones since they're still in the language). A user typically cares about the "easy to abuse and causing trouble when abused" parts. Lots and lots of parts of C++ are like that.

As to the features the FAQ does call evil - why are they in the language? Is it good for the users of the language, or for those who designed and promoted it?

[6.16] Will I sometimes use any so-called "evil" constructs?

FAQ: Of course! Evil means "usually undesirable", but sometimes you have to choose from a set of bad options, and an "evil" feature is your best option. There are no universal rules. Think! At this point the FAQ (and your typical C++ devotee) gets quite agitated.

FQA: Of course! You have no choice. They are built into the language. For example, `"abc"` and `{1,2,3}` are evil arrays, the keyword `this` and the standard `char** argv` are evil pointers, and you'll need an evil `#define` to define a usable interface (for the header file inclusion guards).

Note that with evil arrays, you can write `int a[3] = {1,2,3};` while with the supposedly less evil `std::vector`, you can't. You'll find out that C++ brand new features duplicating the functionality of the "evil" old C features are inferior to the latter in many more ways.

Worse, you can avoid neither the features the FAQ calls evil nor the ones the user would call evil, because if your code doesn't use a feature, it doesn't mean that someone else's code you have to live with doesn't. For example, you may try to avoid exceptions, but the C++ `operator new`, as well as code in third-party libraries, will throw exceptions, and you have to catch them.

There's a basic assumption behind C++ that extra features can't be a problem - only missing features can. That's why there are so many features in C++, and in particular so many duplicate ones. Real world analogies ("imagine a dog with twelve legs") are pale compared to this reality.

[6.17] Is it important to know the technical definition of "good OO"? Of "good class design"?

FAQ: Not if you are a practitioner. Business considerations are the important ones. Precise technical definitions of "good" may lead developers to ignore these considerations, so they are dangerous.

FQA: Whether it's important or not, there is no technical definition of "good", in particular good OO or good class design. "Good" is not a formal term, nor is it universal. For example, if you work for a company, it's important to consider how beneficial something ultimately is for that company in order to define "good".

However, there are technical definitions of OO. So while there are no formal means to tell whether something is *good* OO, you may be able to reason whether something is OO or *not*. Which is not necessarily interesting by itself. But it may be interesting if you have reasons to believe that OO is a good tool for your job - you may want to make sure you'll actually get the benefits you expect. It may also be interesting if someone calls something OO - you may wonder whether you use the same terms or whether they know what they're talking about.

Getting obsessive about precise definitions is a bad way to make decisions. But it's also bad to ignore definitions and blindly go with the hype. For example, people promoting C++ keep telling how good OO is, and how C++ supports OO, and then you try to find out what OO actually *means*, and suddenly it turns out that it's not important. Isn't that a little strange?

It is very beneficial for a practitioner to gain familiarity with OO systems other than C++, and with OO definitions other than the "encapsulation, inheritance, polymorphism" trinity interpreted in special ways allowing C++ to be considered "OO". For example, a claim that an environment lacking boundary checking or garbage collection is not an OO environment sounds outrageous to people accustomed to C++. But from many perspectives, it makes a lot of sense. If anyone can overwrite an object, where's the "encapsulation"? If disposing an object can lead to dangling references or memory leaks, how is the system "object-oriented"? What about the ability to tell what kind of object is located at a given place and time? You say the software works with objects - *where are they*? And if one can't find out, how is one supposed to debug the software?

When people claim that C++ is object-oriented and therefore "good", it may be worth checking whether your notion of "good" is similar to theirs - from a business perspective, for example.

[6.18] What should I tell people who complain that the word "FAQ" is misleading, that it emphasizes the questions rather than the answers, and that we should all start using a different acronym?

FAQ: These people should get a life. Changing a term used and understood by many people is pointless, because people no longer care about the origins of the term and directly associate it with the right meaning.

FQA: If people are accustomed to express an idea in a certain way, and it works for them, trying to convince them to use a new way serves no useful purpose. We could use the opportunity to ask nitpicking questions about how this wisdom is applied to C++ itself. For example, why would someone deprecate `static` variables at the translation unit scope and demand people to use anonymous namespaces to get identical behavior? And all that.

Instead, we'll use the opportunity to point out that at the time of writing (2007), "FQA" appears to be a less popular acronym than "FAQ": a Google search yields a few results, but a Wikipedia search does not. Still, changing "FQA" to something else in this document is not an option: it's all over the place.

Classes and objects

One of the stated goals of C++ is support for object-oriented programming. This page introduces C++ classes and outlines the tactics they use to defeat their purpose.

[7.1] What is a class?

FAQ: In OO software, "the fundamental building block".

A class is a type - a representation for a set of states (much like a C `struct`) and a set of operations for changing the state (moving from one state to another). Classes are similar to built-in types in this sense (for example, an `int` holds a bunch of bits and provides operations like + and *).

FQA: That's a correct theoretical definition. It's equally applicable to all OO languages, but they are different when it comes to more specific, practical aspects of their particular implementation of classes.

How do I create objects? And what happens when they are no longer needed? Is it *my* job to figure out which ones are unused and deallocate them? Bad.

What happens if I have bugs? If I have a pointer to an object, can it be invalid (be a random bit pattern, point to a dead object)? It can? The program will *crash* or worse? What about arrays of objects and out-of-bounds indexes? Crash or a modification of some other random object? You call that encapsulation? Bad.

What happens if I change/add/remove a private value, without changing the interface? All code using the class has to be *recompiled*? I bet you call that encapsulation, too. Bad.

I don't like C++ classes.

[7.2] What is an object?

FAQ: A chunk of memory with certain semantics. The semantics are defined by the class of the object.

FQA: They are also defined by the bugs which cause the code to overwrite data of these objects without bothering to use the interface defined by the class. People who think that real programmers write code without bugs need to upgrade to a human brain.

Still, it sounds interesting: the memory of our C++ program is apparently broken into chunks storing objects of various classes, with "defined semantics". Looks very promising, that. For example, we could ask a debugger about the kind of object located at such a chunk and inspect its data (as in "this is a Point with x=5 and y=6"). We could even take this one step further and implement things like garbage collectors, which can check whether an object is used by looking for pointers to it in the places which are *supposed* to store pointers.

Unfortunately, you can't tell the class of a C++ object given a pointer to it at run time. So if you debug a crashed C++ program and find a pointer somewhere in its guts, and you don't know its type, you'll have to guess that "0000000600000005" is a Point. Which is completely obvious, because that's the way a pair of adjacent integers looks like in hexadecimal memory listings of a little endian 32 bit machine. And two adjacent integers might be a Point. Or some other two-integer structure. Or a part of a three-integer-and-a-float structure. Or they might be two unrelated numbers which just happen to be adjacent.

Which is why you can't automatically collect the garbage of C++ programs.

[7.3] When is an interface "good"?

FAQ: It is good when it hides details, so that the users see a simpler picture. It should also speak the language of the user (a developer, not the customer).

FQA: Um, sometimes you want the interface to expose the many details and speak the language of the machine, although it's probably not very common. The generic answer is something like "an interface is good if it gets the user somewhere".

For example, using OpenGL you can render nifty 3D stuff at real time frame rates. FFTW delivers, well, the Fastest Fourier Transform in the West. With Qt, you can develop cross-platform GUI, and "cross-platform" won't mean "looking like an abandoned student project". Writing that stuff from scratch is lots of work; using the libraries can save lots of work. Apparently learning the interfaces of these libraries is going to pay off for many people.

For a negative example, consider `<algorithm>`. Does `std::for_each` get us anywhere compared to a bare `for` loop, except that now we need to define a functor class? That's a bad interface, because learning it doesn't make it easier to achieve anything useful.

[7.4] What is encapsulation?

FAQ: The prevention of "unauthorized access" to stuff.

The idea is to separate the implementation (which may be changed) from the interface (which is supposed to be stable). Encapsulation will force users to rely on the interface rather than the implementation. That will make changing the implementation cheaper, since the code of the users won't need to be changed.

FQA: That's a nice theoretical definition. Let's talk about practice - the properties of the C++ keywords `private` and `protected`, which actually implement encapsulation.

These keywords will cause the compiler to produce an error message upon access to a non-public member outside of the class. However, they will not cause the compiler to prevent "unauthorized access" by buggy code, for example upon buffer overflow. If you debug a crashed or misbehaving C++ program, forget about encapsulation. There's just one object now: the memory.

As to the cost of changes to the the private parts - they trigger recompilation of all code that `#include`s your class definition. That's typically an order of magnitude more than "code actually using your class", because everything ends up including everything. "The key money-saving insight", as the business-friendly-looking FAQ puts it, is that *every time you change a class definition, you are recompiling the programs using it*. Here's another simple observation: C++ compiles slowly. And what do we get now when we put two and two together? That's right, kids - with C++ classes, the developers get paid primarily to wait for recompilation.

If you want software that is "easy to change", stay away from C++ classes.

[7.5] How does C++ help with the tradeoff of safety vs. usability?

FAQ: In C, stuff is either stored in `struct`s (safety problem - no encapsulation), or it is declared `static` at the file implementing an interface (usability problem - there is no way to have many instances of that data).

With C++ classes, you can have many instances of the data (many objects) *and* encapsulation (non-`public` members).

FQA: This is *wildly wrong*, and the chances that the FAQ author didn't know it are *extremely low*. That's because you can't use `FILE*` from `<stdio.h>` or `HWND` from `<windows.h>` or in fact any widely used and/or decent C library without noticing that the FAQ's claim is wrong.

When you need multiple instances and encapsulation in C, you use a forward declaration of a `struct` in the header file, and define it in the implementation file. That's actually *better* encapsulation than C++ classes - there's still no *run-time* encapsulation (memory can be accidentally/maliciously overwritten), but at least there's *compile-time* encapsulation (you don't have to recompile the code using the interface when you change the implementation).

The fact that a crude C technique for approximating classes is better than the support for classes built into the C++ language is really *shameful*. Apparently so shameful that the FAQ had to distort the facts in an attempt to save face (or else the readers would wonder whether there's any point to C++ classes at all). The FQA hereby declares that it will not go down this path. Therefore, we have to mention this: the forward declaration basically makes it impossible for the calling code to reserve space for the object at compile time. This means that a `struct` declared in a header file or a C++ class can sometimes be allocated more efficiently than a forward-declared `struct`. However, this is really about a *different* tradeoff - safety vs. *efficiency*, and there's no escape from this tradeoff. Either the caller knows about the details such as the size of an object at compile time - which *breaks* compile-time encapsulation - or it doesn't, so it can't handle the allocation.

Anyway, here's the real answer to the original question: C++ helps with the tradeoff of safety vs. usability by eliminating both.

C++ is extremely unsafe because every pointer can be used to modify every piece of memory from any point in code. C++ is extremely unusable due to cryptic syntax, incomprehensible semantics and endless rebuild cycles. Where's your tradeoff now, silly C programmers?

[7.6] How can I prevent other programmers from violating encapsulation by seeing the `private` parts of my class?

FAQ: Don't bother. The fact that a programmer knows about the inner workings of your class isn't a problem. It's a problem if *code* is written to depend on these inner workings.

FQA: That's right. Besides, people can always access the code if a machine can. Preventing people from "seeing" the code in the sense that they can access it, but not understand it is obfuscation, not encapsulation.

[7.7] Is Encapsulation a Security device?

FAQ: No. Encapsulation is about error prevention. Security is about preventing purposeful attacks.

FQA: Depends on the kind of "encapsulation". Some managed environments rely on their support for run time encapsulation, which makes it technically impossible for code to access private parts of objects, to implement security mechanisms. C++ encapsulation evaporates at run time, and is almost non-existent even at compile time - use `#define private public` before including a header file and there's no more encapsulation *(correction)*. It's hardly "encapsulation" at all, so of course it has no security applications - security is harder than encapsulation.

The capital E and S in the question are very amusing. I wonder whether they are a manifestation of Deep Respect for Business Values or Software Engineering; both options are equally hilarious.

[7.8] What's the difference between the keywords `struct` and `class`?

FAQ: By default, `struct` members and base classes are `public`. With `class`, the default is `private`. Never rely on these defaults! Otherwise, `class` and `struct` behave identically.

But the important thing is how developers *feel* about these keywords. `struct` conveys the feeling that its members are supposed to be read and modified by the code using it, and `class` feels like one should use the class methods and not mess with the state directly. This difference is the important one when you decide which keyword to use.

FQA: `struct` is a C keyword. `class` was added to C++ because it is easier than actually making the language object-oriented. And it does a good job when it comes to the *feeling* of a newbie who heard that "OO is good".

Check out the emotional discussion about which keyword should be used in the FAQ. The more similar two duplicate C++ features are, the more heated the argument about the best option to use in each case becomes. Pointers/references, arrays/vectors... Yawn.

By the way, the forward-declaration-of-struct thing works in C++, and it's better than a `class` without `virtual` functions most of the time.

Inline functions

Inline functions are a pet feature of people who think they care about performance, but don't bother to measure it.

[9.1] What's the deal with inline functions?

FAQ: Inlining a function call means that the compiler inserts the code of the function into the calling code (which is technically different, but logically similar to the expansion of `#define` macros). This may improve performance, because the compiler optimizes the callee code in the context of the calling code instead of implementing a function call. However, the performance impact depends on lots of things.

There's more than one way to say that a function should be inline, some of which use the `inline` keyword and some don't. No matter what way you use, the compiler might actually inline the function and it might not - you're just giving it a "hint". Sounds vague? It is - and it is *good*: it lets the compiler generate better and/or more debuggable code.

FQA: To summarize: the compiler has the right to inline or not inline any function, whether it's declared inline in any of the several ways or not. Doesn't this make "inline functions" a meaningless term?

It's impossible to make any sense of this without discussing the history of actual implementations of the C language tool chain - compilers, assemblers and linkers. A straight-forward C implementation (which originally *all* of them were) works like this. First, a compiler generates assembly code from each source file, *separately* (without looking at other source files). Then the assembler converts the assembly code to an "object file", where "object" means "a sequence of bytes" (talk about "object oriented"). For example, a function is one kind of "object" - the bytes encode the machine instructions the compiler used to implement it.

The values of the bytes making up these "objects" are almost completely finalized at this stage. The only kind of "unknowns" is addresses of "objects" - when an "object" refers to an address of another "object" (say, a function calls another function), the assembler can't compute the actual values of the bytes making up the function call instructions. This is done by the linker, which allocates the "objects" (basically by concatenating them). The linker then resolves the references (such as function calls) to the addresses of the "objects".

What this means is that the *only* way to inline functions is to `#include` their definition in the header file - otherwise, the compiler doesn't see the code of the function, and the linker can't do inlining, because all it sees is byte sequences, and it would have to *decompile* them first. Which explains why you need to include the source code of inline functions in header files, but doesn't explain why you need an `inline` keyword and other ways to explicitly declare a function as "inline". After all, the compiler is free to ignore these hints, so what's their point?

Well, the point is that the compiler can't tell an `#include`d function from one written in your source file, because that's how C preprocessing works - the compiler only sees one large file of code. So unless you explicitly declare the `#include`d functions "inline", it will generate their code like it does with normal functions. Then the linker will complain about multiple definitions.

Here's how code is compiled by many modern compilers, including some C and C++ compilers. The compiler transforms the source code to an intermediate representation, "lower" than the source language but "higher" than assembly language. This makes it possible to do inlining at link time, either on a per-library or a whole-program basis. The machine code is only generated as the final linkage step, or it can even be delayed until run time (the so called "just in time compilation"). This way, you don't have to split your functions to "inline" (those that *can* be inlined, but the compiler gets to decide if they actually *are* inlined) and the rest (those that just *can't* be inlined). Instead, you let the compiler make the decision for *all* functions. Unfortunately, the meaning of C and C++ is defined with the old sort of implementation in mind, and having newer, more sophisticated implementations around can't change it.

[9.2] What's a simple example of procedural integration?

FAQ: There's an example of a function calling another function and how inlining may save you copying the parameters when you pass them to a function and copying the result it returns and stuff. There's also a disclaimer saying that it's just an example and many different things can happen.

FQA: Basically, code may be portable, but *performance* is typically not. For example, inlining a function may make code faster or slower, depending on lots of things discussed in the next FAQs. This is one reason to leave the decision to a compiler, because "it understands the target platform better". This is also a reason to leave the decision to a human, because compilers *don't* really know the target platform very well (they know the target processor but not the entire system), and because they don't understand the problem you are solving *at all* (so they can't tell how many times each piece of code is likely to run, etc.).

Anyway, the problem of "helping the compiler to optimize code" by adding "hints" to the code, especially portable code, is quite hard. There are many things similar to inlining in this respect (for example, loop unrolling). Which is why there's no `unroll` keyword forcing loop unrolling. And the `inline` keyword only exists because there's no better way to *enable* (as opposed to "force") inlining in C/C++.

[9.3] Do `inline` functions improve performance?

FAQ: Sometimes they do, sometimes they don't. There's no simple answer.

Inlining can make code faster by eliminating function call overhead, or slower by generating too much code, causing instruction cache misses. It may make it larger by replicating all that callee code, or smaller by saving the instructions used to implement function calls. It may inflate the code of an innermost loop, causing repeated cache misses, or it may improve the locality of reference in the loop, by compiling all relevant code at adjacent addresses. It may also be irrelevant for performance, because your system is not CPU-bound.

See? Told you there was no simple answer.

FQA: However, there *is* a relatively simple answer to the legitimate question: "Why do we need this language feature if its effect is undefined?". See the first FAQ in the section. There's also a relatively simple & useful rule saying that functions which have short code and/or are typically called with compile time constant arguments so that most of their code computes a constant are typically good candidates for inlining. Long functions are typically worse candidates for inlining, because the function call overhead is negligible compared to the things the functions actually do, and the main problem with inlining - large code size - becomes dominant.

It's usually a good idea to only explicitly enable the inlining of very short functions, and performance considerations are not the only reason. In C++ you have to place the code in header files to enable inlining. While the run time performance may or may not improve, the *compile time* performance is guaranteed to drop. Which means changing code becomes hard, which means you'll do your best to *not* change it, which means you'll leave wrong things unfixed. And debuggers handle inlined code pretty poorly (typically you can't inspect the local variables of inlined functions or even step through their source code lines), so debugging the optimized (production) build becomes harder. And debugging a special "debug" build is not always possible (some bugs won't reproduce in that build), not to mention that you have to spend time building those binaries, too.

If your application is not CPU bound, you aren't getting any benefits from using an unsafe language like C++ except for extra quality time with the debugger.

[9.4] How can `inline` functions help with the tradeoff of safety vs. speed?

FAQ: "In straight C" you could implement encapsulation using a `void*`, so that users can't access the underlying data. Instead, the users have to call functions which access that data by casting the `void*` to the right type first.

Not only is this type-unsafe - it's also costly, since the simplest access now involves a function call. In C++ you can use `inline` accessors to `private` data - safe, fast.

FQA: Today C has inline functions (the FAQ probably doesn't consider the current C standard "straight", I wonder why). AFAIK they were back-ported from C++ together with `const` and other useless things. But it's irrelevant to the question, which is about the completely wrong argument that `inline` accessors to `private` functions are a form of "high-speed encapsulation".

First, the tales about `void*` are wrong - you can use forward declarations to achieve the holy grail of (compile time) type safety. Second, a good language implementation can inline the small C-style accessors at link time. Third, `private` provides little encapsulation - change a private member and you have to recompile all code using the class. Fourth, most frequently `private` members with straight-forward `public` accessors are a just verbose way to implement a `public` member, since changing the representation is almost impossible and/or hardly useful.

And in the quite rare cases where it *is* possible and useful, "properties" - a language facility allowing to overload the `obj.member` syntax - could solve the problem, but C++ doesn't have properties. Or you could refactor the code automatically - if anything could reliably parse it.

In the C++ world code is considered a good thing, of which there should be plenty. `private: int _n; public: int n() const { return _n; }` is thus better than `int n;`. The question is - do *you* like lots and lots of C++ code doing practically nothing?

[9.5] Why should I use `inline` functions instead of plain old `#define` macros?

FAQ: Because macros are evil. In particular, when a macro is expanded, the parameters are not evaluated before the expansion, but copied "as is" (they are interpreted as character sequences by the preprocessor, not as C++ expressions). Therefore, if a parameter is an expression with a side effect, such as `i++`, and the macro mentions it several times, the macro will expand to buggy code (for instance, `i` will get incremented many times). Or the generated code may be slow (when you use expressions like `functionTakingAgesToCompute()` as macro arguments).

Besides, inline functions check the argument types.

FQA: Yeah, C macros are no picnic. But see that thing about argument types? How can you write an `inline` function computing the maximal value of two arguments? A `template inline` function, you say? Try this: `std::max(n,5)` with `short n`.

And how should function arguments be passed - by value or by reference? "By value" may cause extra copying, and "by reference" may slow down the code due to aliasing problems, forcing the compiler to actually spill values to memory in order to pass them *to the code of an inlined function*! How's that for "performance benefits"? Another problem C macros don't have.

Frequently `inline` functions are better than macros, though, because the problems with macros turn out to be more severe in many cases. As usual, it's you who gets the interesting job of choosing between duplicate C++ facilities, each flawed in its own unique way.

[9.6] How do you tell the compiler to make a non-member function `inline`?

FAQ: Prepend the `inline` keyword to its prototype, and place the code in a header file, unless it's only used in a single `.cpp` file, or else you'll get errors about "unresolved externals" from the linker.

FQA: The FAQ's decision to avoid the discussion of the *reasons* leading to these requirements is wrong. Clearly people who don't understand the underlying implementation issues won't survive to live the miserable life of competent C++ developers. That's because in C++, the *underlying* stuff tends to climb out of the basement in repeated attempts to make *you* the one lying under a pile of hard, urgent, mind-numbing low-level problems.

Therefore, the only legitimate excuse for telling about a totally weird language requirement and not explaining why it exists is brevity. See the FAQ's lengthy discussion about the performance of inline functions for a pretty good evidence that brevity is hardly the motivation here.

[9.7] How do you tell the compiler to make a member function `inline`?

FAQ: Declare the function in the class as usual. In the definition, add the `inline` keyword to the prototype. The definition must be in a header file.

FQA: Yep, it's similar to non-member functions.

[9.8] Is there another way to tell the compiler to make a member function `inline`?

FAQ: Yes, by writing its code right in the body of the class, instead of only writing the declaration there, and defining it outside of the class. This way, you don't even have to use the `inline` keyword.

It's easier when you write classes, but harder when you read them, because the interface is mixed with the implementation. Remember the "reuse-oriented world"? Think about the welfare of the many, many users of your class!

FQA: What popular language forces you to write a bare interface ("header file") and separately an implementation containing all the information in the interface ("source files")? Somehow all those languages which only make you type the interface once are not at all that hard to use. May it be that it's because these languages are *parsable* and therefore *IDEs can do the oh-so-interesting job of extracting the interface from the implementation* and then you can use things like class view windows to inspect interfaces?

Unless you use templates and operator overloading and all that, many IDEs even have a chance of working with your C++ code. So writing inline functions inside class definitions won't really hurt your users after all. Even if there were no IDEs, you should probably only inline very short functions, with implementations as descriptive as a comment (is `return _x;` less of a documentation than "//returns the value of the x coordinate"?). If there's lots of "implementation details" to hide from the eye of a casual observer, inlining is most likely a bad idea anyway.

[9.9] With inline member functions that are defined outside the class, is it best to put the `inline` keyword next to the declaration within the class body, next to the definition outside the class body, or both?

FAQ: The "best practice" is to only use it in the definition outside of the class. Blah, blah, blah, argues the FAQ passionately about the issue. "Observable semantics", "practical standpoint", blah, blah, blah.

FQA: Programmers typically have a good ability to keep many details in their heads. So good that many don't realize that this ability is *finite*. If you litter your brain with idiotic "best practices" which don't even affect the *observable semantics* of code, you do it at the expense of *not* thinking about something else when you write code. "Something else" may include really important things, like the purpose and the meaning of the code.

If you don't care about this sort of discussions, and you find yourself under an attack of "software professionals" buzzing buzzwords about your "bad practices", send them a link to this page to distract them, and use the time gained by the distraction to go out and buy a buzzer to talk back to them.

References

This page is about C++ references - a duplicate language feature introduced in order to support other duplicate features.

[8.1] What is a reference?

FAQ: It's an alias for an object - another name by which it can be called. The implementation is frequently identical to that of pointers. But don't think of references as pointers - a reference *is* the object.

FQA: A C++ reference is like a C++ pointer except for the following differences:

- You use it as if it were a value: `ref.member`, not `ptr->member`, etc. (in this sense `ref` behaves like `(*ptr)`).
- It must be initialized to point to an object - otherwise, the code won't compile.
- After the initialization, you can't make it point to another object.
- You can't take the address of a reference like you can with pointers (forming a pointer to a pointer).
- There's no "reference arithmetics" (but you can take the address of an object pointed by a reference and do pointer arithmetics on it as in `&obj + 5`).

Strange phrases like "a reference IS the object" are used quite frequently in the C++ community. Such claims are only useful to hide the fact that C++ pointers & references are so similar that having both in the language is an unnecessary complication. In other contexts, the claims are simply false. For example, a wide class of bugs comes from accessing *dangling references* - references to objects which were already destroyed. If a reference is the object, or just another name for it, how can that happen? Names of destroyed objects are inaccessible - it takes a previously assigned pointer to access a destroyed object (C++ also breaks *that* rule - you can access a destroyed global object by its name from a destructor of another global object, but that's a different can of worms).

[8.2] What happens if you assign to a reference?

FAQ: A reference *is* the object, so of course you assign to the referent object.

FQA: Which means that you can't understand the effect of a statement as simple as `a=b;` without knowing whether `a` is a reference or not. A nice feature complementary to references (which make you wonder what "a" means) is operator overloading (which makes you wonder what "=" means). Be careful as you work your way through a quagmire of C++ code.

[8.3] What happens if you return a reference?

FAQ: You can assign to the return value of a function. This is useful for operator overloading, as in `array[index] = value;` where array is an object of a class with overloaded `operator[]` returning a reference.

FQA: Exactly - and *that's* why references exist.

C++ references are essential for supporting C++ operator overloading. That's because C has no facility for assigning to the result of a function call (a function can return a pointer and you can assign to the pointed object, but you need to use an asterisk for dereferencing the pointer, which is different from assigning with the built-in `operator[]`). Some might say that references serve a more generic purpose - they make pointers to objects feel like objects, but for most purposes that can be achieved with `typedef TStruct* T;`.

Operator overloading, in turn, is useful (though not essential) for templates - a duplicate facility solving some of the problems of C macros and creating new, frequently more costly and complicated problems. For example, operator overloading is at the heart of STL - user-defined iterators must have the interface of a C pointer for interoperability with STL algorithms, which can only be achieved with operator overloading.

Operator overloading is also useful for providing user-defined containers such as strings and vectors, duplicating the functionality of built-in strings and arrays.

C++ introduces duplicate facilities in order to introduce other duplicate facilities. Then its apologists try to convince everyone that the old facilities are "evil". Then they explain why removing the old facilities is "impractical".

[8.4] What does `object.method1().method2()` mean?

FAQ: This is called method chaining. `method1` returns a reference to an object of a class having `method2`. This is used in iostream - `cout << a << endl` is method chaining (the method is `operator<<`). A "slick" way to use method chaining is to simulate "named parameters": `a.setWidth(x).setHeight(y)`).

FQA: It's just like `object->method1()->method2()`, but with references instead of pointers.

While method chaining is natural in object-oriented languages, it's good to be aware of problems related to it. At a "high" level of discussion concerned with design, method chaining can be a sign of abstraction violation (for example, do we want the user of `object` to be able to do arbitrary operations with the return value of `method1`?). At a "low" level of discussion concerned with coding, `method1` has no way to report an error except for throwing an exception, which is not always desirable. Inferring a rule like "method chaining is evil" from these issues is probably an exaggeration, but an entire *design* relying on method chaining may raise questions.

For example, iostream is a library for I/O and formatting, and both are a frequent source of run-time errors. How should those errors be checked in statements like `cout << a << b << c ...`? The method chaining used in iostream also makes formatting quite hard - consider the "I/O manipulators".

The "implementation" of named arguments using method chaining is a particularly bad joke.

[8.5] How can you reseat a reference to make it refer to a different object?

FAQ: You can't. The reference *is* the object.

FQA: You can't do it in portable C++. While the reference is probably implemented as a pointer by your compiler, there's no C++ operator to get the address where that pointer is stored. In particular, `&ref` gives the address of the referent object.

One has to work around this extremely rarely, but the need can emerge, especially when the source code of parts of a program is unavailable. If the reference is stored inside an object, you can figure out its offset based on the sizes of other members of the class of the hosting object. You can then modify it as if it were a pointer (as in `*(T**)p = &myobj;`). This is severe abstraction violation, and it isn't portable C++. Make sure you have no other way to achieve your purpose before doing this. In particular, a way involving paying money to the vendor of the code unavailable in source form is frequently a better idea than "clever" hacks like this.

[8.6] When should I use references, and when should I use pointers?

FAQ: Use references unless you can't, especially in interfaces. In particular, references can't point to `NULL`, so you can't have a "sentinel reference". C programmers may dislike the fact that you can't tell whether a modified value is local or a reference to something else. But this is a form of *information hiding*, which is good because you should program in the language of a problem rather than the machine.

FQA: As with most duplicate features, there's no good answer to this question.

C++ programmers use references to denote "a pointer which can't be null and points to a single object rather than an array". Using pointers in these cases confuses people because they assume that pointers are used in the *other* cases. You don't want to confuse people, so you use references to pass arguments to functions. Then it turns out that sometimes you want to pass a null pointer to those functions, and you change references to pointers throughout the code. C++ code ends up containing loads of pointers and references without an apparent reason for the choice made in each particular case.

If you choose to use a reference, make sure you don't need null pointers, pointer arithmetics or reseating. You can't have arrays of references, and you can't store references in container classes, because that would require having uninitialized references and/or pointers to references, and you can't have that. Member references in classes/structs must be initialized in constructors using the ridiculous colon syntax. This makes it harder to reuse initialization code in different constructors, and the problem propagates to the classes using your class as a member since you can't provide a default constructor. There are "smart pointers" (objects with overloaded `->` and `*` operators), but there are no "smart references" (you can't overload the dot), so you won't be able to easily switch to something "smart" later - but that's probably rarely bad since smart pointers are rarely good.

If you choose to use a pointer, make sure you don't confuse other C++ programmers. Pointers to objects of classes with overloaded operators lead to code like `(*parr)[i]`, which gets annoying. In many cases the compiler won't warn you when a pointer is left uninitialized, which may lead to errors harder to debug than simple null pointer dereferencing (not to mention security holes - but there's enough other possibilities for these in a C++ program to make this a separate issue).

These problems mean that in many situations, you must choose between two almost equally bad alternatives. While many people successfully use C pointers *in C*, it doesn't mean that always choosing pointers over references *in C++* produces no new problems - C++ has features, and C++ programmers have habits interacting badly with pointers.

But this, in turn, *doesn't* invalidate the argument "pointers are better than references because it's easy to see whether a side-effect is local or not". *Of course* the behavior of references is "information hiding" - but is it the kind of information you would like to be hidden? Isn't information hiding about making it *easy* to figure out what a program does? How does hiding side effects, which are a very basic cross-cutting semantical aspect of any imperative language, make the code closer to "the language of the problem"?

[8.7] What is a handle to an object? Is it a pointer? Is it a reference? Is it a pointer-to-a-pointer? What is it?

FAQ: A "handle" is something identifying and giving access to an object. The term is meant to be vague, omitting implementation details (a handle can be a pointer or an index into an array or a database key, etc.). Handles are often encapsulated in smart pointer classes.

FQA: One very common way to implement handles in C relies on incomplete types and typedefs:

```
typedef struct FooState* Foo;
```

The definition of `struct FooState` is included in the files implementing `Foo`, but not in the files using it. This is probably the closest thing to "encapsulation" you can get in an unmanaged environment. In particular, it has important advantages compared to C++ classes:

- When the definition of `FooState` is changed, calling code doesn't have to be recompiled (with C++ classes, changing a private member triggers recompilation of user code). This shortens the rebuild cycles and allows to provide stable binary interfaces, simplifying upgrades in many scenarios.
- The module defining `Foo` controls the allocation and deallocation of the objects (with C++ classes, the user is responsible for allocation and must choose between the stack, the global data, the free store and aggregation inside another object). In particular, `FooState` may contain a pointer to the "real" state structure, allowing transparent reallocation of these structures.

The C handle technique is also way better than the "smart pointer" tricks in C++. In addition to readability problems posed by operator overloading (is it a bare pointer or a smart pointer?), and the tedious coding involved, smart pointers provide little encapsulation. That's because overloaded `operator->` must return either a bare pointer or a smart pointer, which means that the *last* smart pointer *must* return a bare pointer, or the compilation will never end (the latter is easy to achieve with templates - like many other not so useful things). So you end up returning a bare pointer to an object of a C++ class, but in a way more convoluted than average C++ code.

When implementing "heavy" classes (unlike, for example, simple objects representing values like points in a 2-dimensional space), using C-style handles is typically much better than pointers or references to objects of C++ classes. Which is quite surprising considering the fact that supporting OO at the language level is one of the main motivations behind C++.

Constructors

This section is about constructors, which create C++ objects, as well as a large number of problems.

[10.1] What's the deal with constructors?

FAQ: A constructor initializes an object given a chunk of memory having arbitrary (undefined) state, the way "init functions" do. It may acquire resource like memory, files, etc. "Ctor" is a common abbreviation.

FQA: That's right - constructors *initialize* objects. In particular, constructors *don't* allocate the chunk of memory used for storing the object. This is done by the code calling a constructor.

The compiler thus has to know the size of a memory chunk needed to store an object of a class (this is the value substituted for `sizeof(MyClass)`) at each point where the class is used. That means knowing all members (public & private). This is a key reason why changing the *private* parts of a C++ class definition requires recompilation of all calling code, effectively making the private members a part of the public interface.

This design of C++ constructors is extremely impractical because the smallest change can trigger the largest recompilation cycle. This would be less of a problem if C++ compiled fast, and if there was an automatic way to detect that code was compiled with an outdated class definition - but it doesn't, and there isn't.

This is one of the many examples where C++ ignores practical considerations in an attempt to achieve theoretical perfection ("language features should *never, ever* impose any performance penalty!"). Compare this to the approach taken in most object-oriented languages, which normally choose true decoupling of interface, sacrificing the efficiency of allocation. Some of these languages provide ways to optimize the allocation of *public* data. This lets you improve performance when you actually need it, but without the illusion of "encapsulation", and have real encapsulation in the rest of your system.

[10.2] Is there any difference between `List x;` and `List x();`?

FAQ: Yes, and it's a *big* one. The first statement declares an object of type `List`, the second declares a function returning an object of type `List`.

FQA: There sure is quite a semantic difference. Too bad it's not accompanied by an equally noticeable syntactic difference. Which is why the question became a frequently asked one.

The problem is that it's hard to tell C++ constructor calls from C/C++ function declarations. The cases discussed here - constructors without arguments - are a relatively moderate manifestation of this problem. All you have to do is memorize a stupid special case: constructor calls look like function calls except when there are no arguments, in which case parentheses must be omitted. There are tons of weird rules in the C++ grammar, and another one doesn't make much difference.

But when there *are* arguments, things get *really* hairy. Consider the statement `A f(B,C);` - this ought to be a function declaration. Why? B and C are surely classes, look at the capital letters. Um, wait, it's just a naming convention. What if they are objects? That makes the statement a constructor call. Let's look up the definition of B and C, that should give us the answer to our question.

Have you ever looked up C++ definitions (manually or using tools such as IDEs)? Check it out, it's fun. You can be sure of one thing: template specialization on integer parameters when the instantiation uses the size of a class defined using multiple inheritance and virtual functions as a parameter value doesn't make definition look-up easy.

When you get an incomprehensible error message from your C++ compiler, be sympathetic. Parsing C++ is a full-blown, industrial strength nightmare.

[10.3] Can one constructor of a class call another constructor of the same class to initialize the `this` object?

FAQ: No. But you can factor out the common code into a private function. In some cases you can use default parameters to "merge" constructors. Calling placement `new` inside a constructor is very bad, although it can "seem" to do the job in some cases.

FQA: You can't - one of the many good reasons to avoid constructors. And anyway, having more than one constructor may be problematic for another reason: they must all have the same name (that of the class), so in fact you're dealing with overloading, and C++ overload resolution is a huge can of worms.

If you're OK with that, you can surely use the FAQ's advice to factor out common code, unless you follow *another* FAQ's advice - the one suggesting to use the ugly colon syntax for initializing member variables. You can't move these things to a function.

If you're in a mood for breaking rules and doing clever tricks, this may be the right time to look for a real problem where such skills are really needed. Breaking C++ rules by calling placement `new` is probably a bad idea - not because C++ rules are any *good*, but because they are pretty *bad* (astonishingly and inconsistently complicated). This is not the kind of rules you can get away with breaking - your maze of hacks is doomed to collapse, and there's enough weight in C++ to bury your entire project when this happens. And anyway, why invest energy into an artificial syntactic problem when there are so many real ones out there?

[10.4] Is the default constructor for `Fred` always `Fred::Fred()`?

FAQ: No, the default constructor is the one that can be called without arguments - either because it takes none or because it defines default values for them.

FQA: Yep, the rule is that the default constructor is the one that can be called "by default" - that is, when the user of a class didn't specify any parameters.

Another neat thing about default constructors is that they get *generated* "by default". For example, if the author of class `Fred` didn't write any constructors for it, a C++ compiler will automatically generate a default constructor for it. You don't see a `Fred::Fred`, but it still exists - invisible C++ functions are great for readability and debugging. The same will happen with other things, for example the *copy constructor*. If you think you can use C++ and simply avoid the features you don't like, you're in for a rude awakening: the invisible compiler-generated stuff is lethal when your class acquires resources (most frequently memory). In for a penny - in for a pound: if you want to write C++ classes, you have to learn pretty much everything about them, and there's plenty.

Where's all this code generated? At the caller's side, of course. To generate it near the rest of the class definition, the C++ compiler would have to know where that definition is, and there's no good way to tell (it may be scattered across as many source files as the author damn pleases). C++ is just like C in this respect - there's no way to narrow the search range for a definition of an entity based on its name and/or namespace; it can be anywhere in the program. In particular, this means that the default constructor will be automatically generated at all points where a class is used, from scratch - yet another reason why C++ code compiles slowly.

And what's needed in order to generate this code? The default constructor generated by default (I hope you follow) calls the default constructors of all class members (which in turn may take auto-generation). Add a private member, and all default constructors generated at the calling code must be regenerated - yet another reason why C++ code must be recompiled frequently.

[10.5] Which constructor gets called when I create an array of `Fred` objects?

FAQ: The default constructor. If there isn't any, your code won't compile. But if you use `std::vector`, you can call any constructor. And you should use `std::vector` anyway since arrays are evil. But sometimes you need them. In that case, you can initialize each element as in `Fred arr[2] = {Fred(3,4), Fred(3,4)};` if you need to call constructors other than the default. And finally you can use placement new - it will take ugly declarations and casts, you should be careful to align the storage right (which can't be done portably), and it's hard to make this exception-safe. See how evil those arrays are? Use `std::vector` - it gets all this complicated stuff right.

FQA: And all you did was asking a simple question! OK, let's start making our way through the pile of rules, exceptions, amendments and excuses.

Given syntax like `MyClass arr[5];` the compiler will generate a loop calling the default constructor of `MyClass`. Pick some time when you have a barf bag handy and look at the generated assembly code. There are two nice things: sometimes compilers will emit a loop even when it's clear that the default constructor has nothing to do; and for automatically generated default constructors, you'll get inline calls to the constructor of each member (for the latter thing you don't really need an array, a single object will do). Bottom line: your program gets bigger and bigger and you don't know why - *you* didn't do anything.

Syntax like `std::vector<Fred> arr(5,Fred(3,4));` gets translated to 5 calls to the copy constructor (in practice - in theory other things could happen; in general, C++ is excellent in theory). This is functionally equivalent to calling `Fred(3,4)` 5 times. The FAQ doesn't mention that it's normally slower though - typically the numbers 3 and 4 must be fetched from the memory allocated for the copied object instead of being passed in registers or inlined into the assembly code.

Of course `std::vector` of a fixed size known at compile time is less efficient than a built-in array anyway. Which is one reason to prefer a built-in array. Initializing such an array with multiple `Fred(3,4)` objects will force you to replicate the initializer, and will produce the biggest possible initialization code. No, the compiler is not likely to notice that you're doing the same thing over and over again, it will repeat the same code N times. On the other hand, if your class does nothing in the constructor but has an "init function", you can call it in a loop. Another reason to avoid constructors.

The horror stories the FAQ tells you about placement new are all true. However, it doesn't mention that `std::vector` always allocates memory on the free store. And there's no standard container you can use for allocating memory on the stack, for example, so if you want one, you'll have to write it yourself. And placement new is necessary to make such containers work with C++ classes without default constructors. Yet another reason to avoid constructors.

Anyway, saying "use `std::vector` because it gets that horrible placement new stuff right" is a little strange if you consider the following assumptions, one of which must apparently hold:

- C++ guys want people to build their own container classes - then why is placement new, which is required to do this right, so hard to use?
- C++ guys don't want people to build their own container classes - then why isn't `std::vector` built into the language, so that things like aggregate initialization could work with it, and maybe we could get rid of those oh-so-evil arrays?

Maybe you can't expect more consistency from the promoters of a language than there is in that language itself.

[10.6] Should my constructors use "initialization lists" or "assignment"?

FAQ: Initialization lists - that's more efficient, except for built-in types (with initializers, you avoid the construction of an "empty" object and the overhead of cleaning it up at the assignment). And some things can't be initialized using assignment, like member references and const members and things without default constructors. So it's best to always use initialization lists even if you don't have to for the sake of "symmetry". There are exceptions to this rule. There's no need for an exhaustive list of them - search your feelings.

FQA: There are good reasons to avoid initialization lists:

- You can't write most kinds of code in the initialization lists - no local variables to reuse results of computations, no loops... Basically there's C++ and there's this initializer land, where you must speak a crippled dialect without any reward for the effort.
- You can't move initializers to a common function to reuse initialization code.
- If you want to pass a pointer to one member to the constructor of another member, you must make sure its constructor is called first, and these constraints can get complicated.
- To help you manage those constraints, some compilers produce annoying warnings when the order of your initializers differs from the order in which you declare the class members, even when it doesn't matter.
- The efficiency argument is wrong for a very common case when the default constructor of a member does nothing.
- Typically compilers attribute all code in initializer lists to the same source code line - for example, the one with the opening brace of the constructor body. When a program crashes inside an initializer, you can't easily tell which one it is using a symbolic debugger.
- Not that it's very different from the rest of C++, but the colon syntax is just plain ugly.

The trouble with initialization lists, like with any duplicate language feature, is that you *can't* really avoid using it. For example, there are classes without default constructors, or with relatively heavy ones. So "never use initialization lists" is not a very useful rule, and the best answer, as usual, is that there's no good answer.

However, initialization lists are really less than useful, so it's probably good to avoid them unless you can't. In particular, avoiding reference and const members and having lightweight default constructors in your classes may help.

[10.7] Should you use the `this` pointer in the constructor?

FAQ: Some people think you can't since the object is not fully initialized, but you can if you know the rules. For example, you can always access members inside the constructor body (after the opening brace). But you can never access members of a derived class by calling virtual functions (the function call will invoke the implementation of the class defining the constructor, not that of the derived class). Sometimes you can use a member to initialize another member in an initializer list and sometimes you can't - you must know the order in which members are initialized.

FQA: That's right - all problems and questionable scenarios come from tricky C++ things, like initialization lists and virtual function calls from constructors. If you avoid initialization lists and use plain old assignment or initialization function calls in the constructor body, you can be sure all members can be used - you have one problem solved. Use init functions instead of constructors and virtual function calls will invoke functions of the derived class - another problem solved.

An alternative to avoiding the dark corners of C++ is to spend time learning them. For example, you can memorize the rules defining the order of initialization, and rely on them heavily in your code. That way people who need to understand and/or maintain your code will have to get pretty good at these things, too, so you won't end up being the only one around knowing tons of useless obscure stuff. Or they will have to ask you, which increases your weight in the organization. You win big either way, at least as long as nobody chooses the path of physical aggression to deal with the problems you create.

[10.8] What is the "Named Constructor Idiom"?

FAQ: It's when you have static functions ("named constructors") returning objects in your class. For example, this way you can have two "constructors" returning 2D point objects: one getting rectangular coordinates (2 `float`s), and another getting polar coordinates (also 2 `float`s). With C++ constructors you couldn't do it because they all have the same name, so that would be ambiguous overloading. And this can be as *fast* as regular constructors! And you can use a similar technique to enforce an allocation policy, like having all objects allocated with `new`.

FQA: Three cheers! This is almost like C init functions - a step in the right direction. Namely, this way we don't have overloading problems, and we can get rid of the pesky separation of allocation from initialization - client code doesn't really have to know about our private members anymore.

The only thing left to do is to ditch the whole class thing, and use C pointers to incomplete types as object handles - that way we can actually modify private members without recompiling the calling code. Or we can use a real object-oriented language, where the problem doesn't exist in the first place.

[10.9] Does return-by-value mean extra copies and extra overhead?

FAQ: "Not necessarily". A truly exhausting, though not necessarily exhaustive list of examples follows, with many stories about "virtually all commercial-grade compilers" doing clever things.

FQA: Let's enumerate the possibilities using boring binary logic:

- You care about performance.
- You don't care about performance.

In the first case, "not necessarily" is not a good answer for you. You don't want your code to be littered with things like return-by-value and later wonder why your "commercial-grade" compiler emits huge and slow code in some of the performance-critical places. If performance is one of your goals, you're better off writing code in ways making it as easy as possible to predict performance, without knowing an infinite amount of details specific for each of the compilers you use for production code. And if you have experience with optimization, you've probably noticed another thing - most C++ compilers work hard on optimizing the C subset, but are pretty dumb when it comes to C++-specific parts. Compiler writers are probably happy if they can correctly *parse* C++ code and somehow lower it to C and have the back-end optimize stuff at the C level. Basically to get performance you need C, because that's what today's optimizers are best at, and you're wasting time with C++.

If you don't care about performance, you are *also* wasting time using C++. There are hordes of programming languages designed for managed environments, so you won't have problems coming from undefined behavior of all kinds. And the vast majority of languages are way simpler and more consistent than C++, so you'll get another huge burden off your back.

Of course our boring binary logic fails to represent the developers who *think* they care about performance, although they have a very vague idea about the actual performance of their programs. Those are the people who use the von Neumann model ("you access memory using pointers which are actually indexes of individual bytes") to think about computers, and call this "the low level". They are typically less aware of things like instruction caches (which make *big* programs *slow*), SIMD instruction sets (which can give performance gains *way beyond* "generic" template implementations of numerical algorithms), assembly language and optimizers in general (for example, how `restrict` helps optimization but `const` doesn't). These people engage in lengthy discussions about complicated high-level optimizations, their ultimate goal being very generic code which can be compiled to a very efficient program by a very smart compiler, which will never exist. These people are welcome to waste as much time thinking about return-by-value optimizations as they wish, as long as it prevents them from causing actual damage.

[10.10] Why can't I initialize my `static` member data in my constructor's initialization list?

FAQ: Because you must define such data explicitly as in `static MyClass::g_myNum = 5;`.

FQA: Because it's meaningless. `static` members are global variables with respect to allocation and life cycle, the only difference is in the name look-up and access control. So they are instantiated and initialized once per program run. Initialization lists initialize the members of objects which are instantiated with each object, which can happen more or less times than once per program run.

One possible reason making this question frequently asked is that you can *assign* to static variables in the body of a constructor, as in `g_numObjs++`. People trying to follow the advice to use the crippled C++ subset available in the initialization lists might attempt to translate this statement to initializer-like `g_numObjs(g_numObjs + 1)` or something, which doesn't work.

You can probably look at it both ways - "initializers are used to initialize things instantiated per object" and "the subset of C++ available in initialization lists makes it impossible to do almost anything".

[10.11] Why are classes with `static` data members getting linker errors?

FAQ: Because you must define such data explicitly as in `static MyClass::g_myNum = 5;`.

FQA: Beyond being annoying, this is quite weird. At the first glance it looks reasonable: after all, C++ is just a thick layer of syntax on top of C, but the basic simpleton mechanisms are the same. For instance, definition look-up is still done using header files holding random declarations and object files holding arbitrary definitions - each function can be declared anywhere (N times) and defined anywhere (1 time).

So the compiler can't let you define something which becomes a global variable at the C/assembly level in a header file as in `static int g_myNum = 5;` - that way, you'd get multiple definitions (at each file where the class definition is included). Consequently, the C++ syntax is defined in a way forcing you to solve the compiler's problem by choosing a source file and stuffing the definition there (most frequently the choice is trivial since a class is implemented in a single source file, but this is a *convention*, not a rule the compiler can use to simplify definition look-up).

While this explanation doesn't make the syntax and the weird "undefined external" errors any nicer, at least it seems to make sense. Until you realize that there are *tons* of definitions compiled to C/assembly globals in C++ that *must* be placed at header files. Consider virtual function tables and template classes. These definitions are compiled over and over again each time a header file gets included, and then the linker must throw away N-1 copies and keep one (if the copies are different because of different compiler settings and/or preprocessor flags, it's your problem - it won't bother to check).

It turns out that C++ *can't* be implemented on top of any linker that supports C - the linker must support the "generate N times and throw away N-1 copies" feature (or is it a documented bug?), instead of issuing a "multiple definition" error. In GNU linkers this is called "linkonce" or something. To support C++, you must add features to the linker. Too bad they didn't think of adding type-safe linkage (checking the consistency of definitions used in different object files) while they were at it.

The conclusion is that there is no technical reason whatsoever, even inside the twisted C++ universe, to make the syntax harder for the user in this case. On the other hand, there's no reason to make the syntax *easier* either. That would just introduce inconsistency with the rest of the language.

[10.12] What's the "static initialization order fiasco"?

FAQ: A subtle, frequently misunderstood source of errors, which are hard to catch because they occur before `main` is called. The errors can happen when the constructor of a global object defined in `x.cpp` accesses (directly or indirectly) a global object defined in `y.cpp`. The order of initialization of these objects is undefined, so you'll see the problem in 50% of the cases (an error may be triggered by a rebuild). "It's that simple".

FQA: And it's *that* stupid. Just look at this:

- C++ doesn't define the order of initialization of global objects.
- C++ lets you write code that depends on the order of initialization of global objects.
- C++ doesn't have a mechanism to automatically detect these errors, neither at compile time nor at run time.
- C++ doesn't make the slightest effort to help you debug the problem (for example, go figure in which actual order the objects were initialized).

With most duplicate language features, one can't simply say "avoid the new broken C++ features" because the language works so hard to get you in trouble if you do. But in this particular case it's probably a good rule. Consider the reasons to avoid instantiating global objects with non-trivial constructors (ones that do more than nothing), and instead use plain old C aggregate initialization:

- You have to worry about this "fiasco" business, which is quite nasty. With C initializers, once you passed compilation and linkage, there are no problems - the linker creates a snapshot of the initial state of your global variables, and there are no failure modes.
- You have all this initialization code before `main`, with corresponding clean-up code called after `main`. What if you want to clean up and re-initialize things *without* exiting `main`? People calling `getenv` or accessing the command line using platform-specific tricks in code called before `main` further improve the life of other people working on the project.
- Your program contains a huge bulk of slow initialization code. A C statement like `Point p={4,5};` gets translated to 8 bytes representing the integers 4 and 5 in the executable. A C++ statement like `Point p(4,5);` gets translated to *code* storing 4 and 5 to `p`. What are the benefits except for getting large programs starting up slowly?

Some people believe that you need non-trivial global constructors, or else you'll have to make your user call an initialization function to work with your module. The fact is that the experienced users *prefer* to initialize modules explicitly rather than having them autonomously kicking in, possibly crashing the program because of dependency issues, not to mention printing messages and popping up configuration dialogs. All of these implicit things tend to become quite *explicit* on the day when you least need them to. Quoting the FAQ's favorite expression, just say no.

[10.13] How do I prevent the "static initialization order fiasco"?

FAQ: By using the "Construct On First Use" idiom, as in

```
MyClass& getMyObj()
{
  static MyClass* p = new MyClass;
  return *p;
}
```

This solution may "leak". There's another solution working around this problem, but it creates other problems.

FQA: By not instantiating global objects with constructors doing more than nothing. That prevents this and other kinds of "fiasco". The technique in the FAQ, frequently referred to as an implementation of "the singleton design pattern" (BTW, you can put quotes wherever you like in that one, for example the "singleton" "design" "pattern"), has the following problems:

- As mentioned by the FAQ, it leaks memory *and possibly other resources* (C++ people love to talk about how memory is "just one kind of resource", overlooking the fact that it accounts for 99% of "resources" actually used by a program, so special, "non-generic" things like garbage collection may be appropriate to manage it - but they suddenly forget all about it when it comes to this "construct on first use" business). Practitioners may find it intolerable to have these memory leaks reported by tools designed to detect leaks, even if their platform is capable of disposing the garbage left by a program automatically. People interested in a more theoretical discussion may enjoy the recommendation to violate the C++ rules regarding memory management found in a "standard practice".
- It is slow, not to mention thread safety. When you have a plain C global initialized by a plain C explicit initialization function, accessing it is quite easy (function arguments may be faster to fetch, or they may be not). In this case, you call a function (which can be inlined - that doesn't necessarily help though) that contains a hidden `if` statement. You don't see it? Well, how do you think the clever function manages to only call `new` once? There's a global boolean (sometimes called "guard variable") which it checks each time. To access a global, you now need to access two globals. If you find it interesting, you can list the symbols of your program (using a program like `nm`), and then filter the output using a C++ name demangler (like `c++filt`). You may find lots of "guard variable of something" kind of names there.

The C++ global initialization and destruction support is broken. As usual, the "idiom" in the FAQ "solves" the problem by creating worse problems.

[10.14] Why doesn't the construct-on-first-use idiom use a static object instead of a static pointer?

FAQ: It solves one problem and creates another problem. Namely, you avoid the resource leak, but your program can crash because now you achieved "construct-on-first-use", but not "destroy-after-last-use", so someone can access a dead object (using `new` means "never destroy", which is at least guaranteed to be *after* last use).

Actually there's a *third* approach solving *both* problems (initialization and destruction order), having "non-trivial cost". The FAQ author feels "too lazy and busy" to explain - go buy his "C++ FAQ Book" to read about it.

FQA: The FAQ is right - this variant, known as "the Meyers' singleton", is also broken. And it's probably the hardest to work around, too. At least with the `new` variant you *can* delete the object with `delete &getMyObj();` - increasingly ugly, but it may yield a working (though unmaintainable) program. With the static variable technique, C++ records a pointer to the destructor in some global array, so you won't be able to control the order of destruction (it's always the order of construction, reversed - and you *didn't* want to control the order of construction, you wanted it to "just work", right?).

As to other approaches - I do have enough energy and spare time to repeat a *free* advice: use explicit initialization and clean-up functions, and realize that the initialization and destruction sequences are part of the *design* of your modules and your program.

If it's hard to get out of the state of mind where you think initialization should somehow work out by itself, as opposed to the really interesting things your program does afterwards (the "steady state"), maybe an analogy with hardware can help. In a simplified model of synchronous hardware design, you have two "trees" of signals (basically wires) reaching almost every place in your system: the *clock* (which synchronizes things done at the steady state), and the *reset* (upon which all the initialization is conditioned). A common milestone in hardware design is getting the reset right. When you buy hardware, a large portion of it is devoted to initialization (which is why you can *reboot* it and it enters a reasonable state - you can't do that with biological "computers" like the human brain).

By the way, I didn't read the C++ FAQ book (I really don't have time for a C++ FQA book at the moment, so why bother?). But I did read "Modern C++ Design", which also offers a solution. So if you buy the FAQ book and find the third "solution" and it involves having your singletons instantiated from a hairy template with a global data structure keeping "priorities" or some such and registering `atexit` callbacks which can in turn call `atexit` which detects dark areas in the C and C++ standards because nobody thought anyone would ever do that - if that's the third way, be sure that the cost is *really* "non-trivial".

[10.15] How do I prevent the "`static` initialization order fiasco" for my `static` data members?

FAQ: Using the same techniques just described, except for using static member functions instead of global functions. For some reason a long list of examples follows, as well as a discussion on performance.

FQA: There's no difference between static data members and global variables except for name look-up and access control. So you can either use the broken & slow techniques from the FAQ, or you can avoid non-trivial constructors and live happily ever after.

There's no point in having separate discussions on static data members and plain global variables. Well, except for mentioning that static data members are a brand new C++ feature, and it's particularly nice that C++ recommends to wrap its own syntax with another layer of its syntax for "safety". An alternative approach to safety is to use zero layers of C++ syntax.

[10.16] Do I need to worry about the "`static` initialization order fiasco" for variables of built-in/intrinsic types?

FAQ: Yes, if you use function calls to initialize them as in `int g = f();` - that way `f` can access `g`, or you can have other dependency problems.

FQA: Exactly - and this code doesn't compile in C. We seem to have a pretty clear picture here, don't we? As a rule of thumb, the answer to the more general version of the question - "Do I need to worry when I use a C++ feature not available in C?" - is also "Yes".

Not that plain C is very safe, mind you. If you don't need the performance, you can always switch to a safer, higher-level language. But at least C doesn't pretend to be very high-level, and makes less promises it can't keep (like "go ahead, use whatever code you like to initialize global variables - see how high-level our language is?").

[10.17] How can I handle a constructor that fails?

FAQ: Throw an exception.

FQA: Right.

Q: What do you do when a ship is on fire?
A: Drown it. The fire will stop immediately.

Seriously, C++ exceptions are a leading candidate for the title "the worst way to handle run time errors ever invented". But constructors can't return values. Even though they *don't* technically return an object - they merely initialize a chunk of memory they are passed. So they *could* return status information despite the C/C++ limitation of at most one return value per function. Which itself has no technical justification. This is yet another reason to avoid constructors that do more than nothing. It's also yet another illustration how hard it is to use only some of C++ features and avoid others ("we want classes, but we don't need exceptions" - until you want to handle an error in a constructor).

It is also notable that the C++ standard library *doesn't* handle errors in constructors (or overloaded operators, which pose the same problem) using exceptions (for example, consider `ifstream` and `ofstream`).

[10.18] What is the "Named Parameter Idiom"?

FAQ: A useful application of method chaining. It works around the fact that C & C++ don't have keyword arguments, and it does that better than combining the parameters in a string or a bit mask. It works like this:

```
File f = OpenFile("language-abuse.txt")
         .useWeirdLineBreakRules(true)
         .writeLotsOfGoryDetails(true)
         .averageNumberOfExcuses(ZILLION);
/* a sizable implementation with two classes and friends
   and methods returning *this and what-not omitted */
```

And if method calls are inlined, there's no speed overhead (but the code size may "slightly" increase - but that's a long story).

FQA: Let's have a closer look at this Named Parameter Idiocy.

- The implementation in the FAQ pushes these parameters into a temporary structure, which is normally slower than argument passing.
- And the multiple method calls, inlined or not, are no picnic either. Compare assembly code generated by `printf` and `iostream` calls if you doubt. This way you can compare your notion of "slightly" with that used by the designers of C++.
- All ways to implement this require the author of the class to write a lot of code ultimately doing nothing. Which is OK

because in theory, each class is written once and used thousands of times. In practice most classes keep changing forever and are used in a couple of places. Which doesn't matter as long as you only use C++ in theory where it belongs.
- But that would be justified to an extent if it made things easier for the user. But it doesn't. Try reading the header file with the `class OpenFile` and figuring out how to open files.
- And now that we have this `OpenFile` class, people will start storing objects of this class and passing them around. Which could be a good idea in some cases, except that the class has a stupid function-like name, further enhancing readability.
- Or we could use a different implementation, chaining the methods of `class File` and avoiding the second class. But this way, we can't detect errors easily, because there's no point where we see all of the parameters at once, we get them one by one.

This syntactic sugar of the bitter kind raises two questions. First, why don't we have keyword arguments in C++? They are much easier to implement in a compiler than virtually any feature C++ added to C, and are way more useful, too. And second, if for some reason you have to work in C++, what's the problem with accepting the fact that it doesn't have keyword arguments, and using a structure of parameters in way at least making it *clear* that you use a structure of parameters? For example:

```
OpenFileParams params;
params.useNormalLineBreaks(true);
params.quitFiddlingWithSyntax(true);
File file; //trivial constructor
if(!file.open("grow-up.txt",params)) {
  //handle errors without the "help" of C++ exceptions
}
```

This doesn't solve the problems with code speed & size, but at least the code of the class and the code using the class are reasonably readable and writable. And there seem to be no new problems, unless someone considers the fact that our code is now pretty close to C and we no longer rely on C++ method chaining a problem.

Abusing C++ syntax in order to cover up deficiencies of C++ syntax, thus creating real problems in attempts to solve non-problems, is a popular hobby among C++ professionals. Let's check if it makes sense by imagining that someone spent the better part of the day writing all this wrapper code doing trivial things. Let's try to help that someone explain what was accomplished to someone else, and further assume that the other someone has a mind not entirely buried in C++ quirks. "I was writing this... Um, you see, I was opening a file... I wrote this interface of 2 screens of code... And then..." Sheesh, that's embarrassing.

[10.19] Why am I getting an error after declaring a `Foo` object via `Foo x(Bar())`?

FAQ: This *hurts. Sit down*.

The "simple" explanation: this doesn't really declare an object; `Foo x = Foo(Bar());` does. The complete explanation for "those caring about their professional future": this actually declares a function named `x` returning a `Foo`; the single argument of this `x` function is of type "a function with no argument returning a `Bar`".

Don't get all excited about it. Don't "extrapolate from the obscure to the common". Don't "worship consistency". That's not "wise". The FAQ actually says all these quoted things.

FQA: Those who use computers to do any useful work are probably immune to brain-crippled syntax, because there are more painful things, like brain-crippled semantics. So sitting down is only necessary if you find it comfortable. For similar reasons, don't worry about your professional future too much if this looks boring and stupid and you don't feel like understanding it. It really is what you think it is, and programming is not supposed to be about this kind of thing. If someone rejects you in an interview because you don't know the answer to a question like this, you are lucky - you've just escaped a horrible working environment.

Anyway, if for some reason you *are* curious to find out why this happens, it turns out the FAQ has no answers, it just presents it as an arbitrary rule. It doesn't help to understand anything. The statement is ambiguous - you can read it both as a constructor call and a declaration of a peculiar function. How does the compiler know? Let's try to see.

Apparently the key problem is that in order to tell a function declaration from a constructor call, you need to know whether the things in the parentheses are objects or types. This makes parsing `A f(B);` non-trivial. In our example, things are complicated by the fact that `Bar()` itself presents *the very same problem* - is it a function declaration or a constructor call? Worse, this time the darn thing accepts no arguments, so you can't use them to figure it out. Well, C++ has an arbitrary rule to help the compiler (but not necessarily the user): things with empty parentheses are function declarations, unless that's entirely impossible (list of special cases follows). That's why `A x();` declares a function but `A x;` defines an object. And that's why `Bar()` is interpreted as a function declaration, which means that it's an argument type, not an object, so the whole statement is actually a function declaration. I think.

The problem with `Foo` and `Bar` is that the C++ grammar is FUBAR. In a language designed in a more orderly way (which is most of them), there are no such ambiguities. A relatively simple way to make sure there are none is to use a formal grammar specification and feed it to a program that generates a parser for the language, checking the consistency of the grammar on the way. `yacc/bison` is one mature program of this kind; there are newer ones with more features, which can represent more complicated grammars (but AFAIK no generic tool is capable of representing the extremely complicated, inconsistent and actually undecidable C++ grammar).

It could help the users if the people promoting C++ realized that the consistency of a grammar is not something you "worship", but a technical property which is far easier to achieve than it is to deal with the consequences of not having it. This example is just one drop in the ocean of such problems. The complexity of the C++ grammar guarantees that compiler error messages will remain cryptic, debuggers will stay ignorant, and IDEs will be unhelpful compared to those available for other languages forever.

Destructors

Destructors are one of the many pieces of the puzzle that is the C++ memory management.

[11.1] What's the deal with destructors?

FAQ: A destructor, abbreviated "dtor", is a "you're about to die" member function. It is used to release resources, for example, semaphores. Most frequently the resource is memory allocated by `new`; the destructor frees it with `delete`.

FQA: Yep, C++ has no garbage collection, which makes "memory" a manually managed "resource". Die-hard C++ programmers firmly believe that RAII - acquiring resources in constructors and releasing them in destructors - is the silver bullet for resource management. The problem with this approach is that many objects can point to the same piece of data, but only one of them is the "owner", and when the owner dies, its destructor promptly blows the piece of data to even smaller little pieces.

The most common way around this is *copying*. For example, you load a large file representing a 3D object model or an XML document or something. You find an interesting leaf on the 3D tree model or in the XML tree or whatever. Now you want to release the memory of the whole model so that you can free some space for processing the part. You need to either convince the model object that it doesn't own the stuff in your leaf, or you have to copy that stuff. The latter is way easier (try to convince an owner such as `std::list` to let a couple of nodes go out of its jurisdiction).

Which is one reason that can cause garbage collection to outperform manual memory management. But of course with garbage collection you'll never master your debugging tools to an extent anywhere near the expertise you'll reach with manual memory management (you'll have 10 times less bugs to practice on). It's up to you to choose the right tradeoff.

There are two common counter arguments: not all resources are memory, and not all systems can afford garbage collection. Well, surely more than 95% of resources *are* memory, and surely more than 95% of C++ code runs in systems which *can* afford garbage collection. These arguments are yet another evidence that C++ is about theory, not practice. And the resources which are not memory and must be released deterministically usually aren't handled very well be C++ destructors anyway. That's because a C++ destructor can't handle errors (it has no return value and it can't throw exceptions).

[11.2] What's the order that local objects are destructed?

FAQ: In reverse order of construction - the stuff declared last is destroyed first.

FQA: Which makes sense, but are you sure you want to write code that depends on this?

[11.3] What's the order that objects in an array are destructed?

FAQ: In reverse order of construction - the last element is destroyed first.

FQA: Which sort of makes sense, but you *surely* don't want to write code that depends on that one, do you?

By the way, don't forget to use `delete[]`, not just `delete`, to free arrays allocated with `new[]`, otherwise the stupid thing will not bother to check the number of elements pointed by your pointer and some of your destructors won't get called (actually, in theory it could be worse - the behavior is undefined).

[11.4] Can I overload the destructor for my class?

FAQ: No. Destructors never have parameters or return values. And you're not supposed to call destructors explicitly, so you couldn't use parameters or return values anyway.

FQA: Why do you want to overload destructors? Do you like C++ overloading? Are you sure? But the lack of return values *is* a pity - no way to handle errors. Let's hope they won't happen, shall we?

[11.5] Should I explicitly call a destructor on a local variable?

FAQ: Don't do that! The destructor will be called again at the end of scope. And the second call will do nasty things, like crashing your program or something.

FQA: There are standard questions to ask in these cases, like "if this syntax is never useful, why does it compile"? Anyway, you *can* call a destructor on a local variable, but you have to make sure you call a constructor after that and before the object goes out of scope. For example:

```
AMiserableFileClass file("f1.txt");
//use file... now we want to close it, but there's no close() method, so:
file.~AMiserableFileClass();
//open another file
new (&file) AMiserableFileClass("f2.txt");
```

`AMiserableFileClass` is a miserable file class because it has no `close` method, so you might feel the need to close it in the ugly way above. Try to avoid this, because most people won't understand it, and they shouldn't, because there are better things to do than fiddling with the many ugly bits of C++.

[11.6] What if I want a local to "die" before the close `}` of the scope in which it was created? Can I call a destructor on a local if I *really* want to?

FAQ: No, no, no! See the next FAQ for a simple solution. But don't call the destructor!

FQA: If you ever get into the situation of promoting an especially disgusting product, such as the C++ programming language, there are better ways to handle it than get all excited. You can try and find a less disgusting product to center your money-making around, or you can just relax.

"Don't call the destructor". There has to be *some* reason for the destructor call to compile, doesn't it? Perhaps sharing it with us could help calm down.

[11.7] OK, OK already; I won't explicitly call the destructor of a local; but how do I handle the above situation?

FAQ: Now you're talking! Here's a tip - you can simply surround the object with an *artificial block*:

```
{
  AMiserableFileClass file("f1.txt");
} //the file object dies here
```

FQA: Is this ugly code supposed to be better than the ugly code calling a destructor and than a constructor? On a level, this version is more cryptic (at least it's easy to see what gets called when in that other ugly piece of code). But the truth is that for many actual C++ programmers the "artificial blocks" (isn't all code "artificial"?) are more readable.

[11.8] What if I can't wrap the local in an artificial block?

FAQ: If you absolutely have to, do something like adding a `close` method to `AMiserableFileClass` which closes the file in its destructor. So you can achieve the effect of the destructor call without calling the destructor. Which is a taboo, get it?

The FAQ also points out how hard it is to write a `close` function so that the destructor doesn't try to close a closed file.

FQA: If you can change `AMiserableFileClass`, it's better than using ugly code to work around its deficiencies. But where's the FAQ's brave new reuse-oriented world now? What about all those classes with stable interfaces you can't change? Seriously, it can happen with classes not available in source form.

I sincerely believe that the not-so-appetizing "call the destructor, then call a constructor" method is legal C++. If this is indeed so, I fail to understand the problem with mentioning it in the answer.

As to the problem of having destructors and close functions respect each other - one way around this is to avoid non-trivial constructors and destructors and always use init and close functions. Next, you can replace C++ classes with their metaphysical "encapsulation" with forward-declared C `struct`s, which can actually yield stable binary interfaces and save recompilation. Next, you can replace C++ with C (where execution time matters) or with one of the many sane, safe languages (where development time matters). There, doesn't it feel much better now?

[11.9] But can I explicitly call a destructor if I've allocated my object with `new`?

FAQ: You can't, unless the object was allocated with placement `new`. Objects created by `new` must be `delete`d, which does two things (remember them): calls the destructor, then frees the memory.

FQA: Translation: `delete` *is* a way to explictly call a destructor, but it *also* deallocates the memory. You can also call a destructor without deallocating the memory. It's ugly and useless in most cases, but you can do that.

Questions like "What *exactly* does `delete` do?" probably cross the fine line separating between the questions everyone claiming to know C++ should be able to answer and the questions identifying people with too much C++-related garbage in their brains. People can be quite productive by simply calling `new` and `delete` without thinking too much about the two steps involved, so not knowing about the two steps is probably no big deal. The most interesting thing about the two steps is that the idea of having the code using the class managing the memory of its objects is the main reason C++ code is recompiled so frequently.

[11.10] What is "placement `new`" and why would I use it?

FAQ: There are many uses for placement `new`. For example, you can allocate an object in a particular location, you can pass the pointer to this location to placement `new` like this: `C* p = new(place) C(args);`

Don't use this unless you really care where the object lives (say, you have a memory-mapped hardware device like a timer, and you want to have a `Timer` object at the particular location).

Beware! It is your job to make sure that the address where you allocate the object is properly aligned and that you have enough space for the object. You are also responsible for calling the destructor with `p->~C();`.

FQA: Size is relatively easy to deal with because we have `sizeof`, alignment can be more painful. Here's an advice: if you care where your objects are allocated, don't make them objects of classes with constructors and destructors and stuff. This way, you can create memory pools with code like `C pool[N];`, which takes care of alignment *and* "type safety". But if the class `C` has a constructor, it's going to get called N times by this statement (slow and stupid), and if you want someone to use placement `new` with the pool, you'll have to *call the destructors* after the constructors finish running (slow, stupid and cryptic). Or you can use `char pool[N*sizeof(C);]` with platform-specific additions to handle alignment, and then you won't be able to easily inspect the pool in a debugger (the object `pool` has the wrong type), etc.

And you have to be *out of your mind* to call placement `new` when you deal with memory-mapped hardware. Do you realize that the constructor is going to *directly modify the state of the hardware*? Even if you get this right (yes, there are ways to get this wrong, too boring to enumerate here), this is one very unmaintainable way to write this kind of code. If you think that's "intuitive", think again. What is the constructor doing - "creating" the timer? Come on, it's *hardware*, it was already physically there. Oh, it "initializes" it? So why don't use a function with "init" in its name instead of a "constructor"?

You have enough trouble thinking about the semantics of the hardware interface, so why would anyone want to add the complexity of C++ to the problem?

At least the FAQ has finally disclosed the top secret information about explicitly calling destructors.

[11.11] When I write a destructor, do I need to explicitly call the destructors for my member objects?

FAQ: No, they are called implicitly in the reverse order of their declaration in the class.

FQA: Pay attention to the details. If your member is a pointer to something allocated with `new`, *the pointer's destructor*, which is a purely metaphysical entity doing nothing physically observable, is called, but it's *your* job to call `delete` on the pointer, which calls the destructor of the pointed object. The pointed object is technically *not* a member of your class (the pointer is). The difference between the intuitive feeling that "it's part of the class" and the formal definition of the term "member" is one reason making `const` less than useful (is changing the object pointed by a member "changes" the object itself or not?).

Experienced C++ programmers find this natural since the C++ approach to these issues is relatively uniform. If you have the tough luck of using C++ a lot, this point is obvious (but you wouldn't ask the question in the first place).

There's a nifty thing called "garbage collection" that's been around for nearly half a century. The run time system automatically figures out which objects are not pointed by anything and collects this garbage. Check it out, it's quite nice.

[11.12] When I write a derived class's destructor, do I need to explicitly call the destructor for my base class?

FAQ: No, this is done implicitly, in the reverse order of the appearance of the classes in the inheritance list (but `virtual` inheritance complicates matters), and after the destruction of the class members.

FQA: The FAQ says that if someone is relying on more complicated details of the finalization order, you'll need information outside the scope of the FAQ. The remark probably refers to useful information like locations of mental institutions and drug prescriptions.

Seriously, a Usenet language FAQ is already a place normally visited only by language lawyers. If you need *more* obscure information to get your job done, the only legitimate reason is that you're writing a C++ compiler. If that's the case, and you feel miserable, cheer up - it could be worse. For example, imagine the misery of the people who'll *use* your compiler!

[11.13] Should my destructor throw an exception when it detects a problem?

FAQ: THAT IS DANGEROUS! Which is discussed in a FAQ about exceptions.

FQA: It shouldn't. And you can't return an error code either. However, on many systems you can send an e-mail with the word "BUMMER" to your support e-mail address.

If you want your destructor to detect problems, make it a `close` function.

[11.14] Is there a way to force `new` to allocate memory from a specific memory area?

FAQ: Oh, yessssss! Pools! Pools, pools, pools, pools, pools...

Please forgive me this time. I can't summarize what the FAQ is saying. It's almost as long as *all the previous answers*. And it's totally out of control. If you feel like it, fasten your seat belt, grab a barf bag, follow the link and knock yourself out.

FQA: Oh, nooooooo! Don't do that!

I know that your system has more than one kind of memory and/or you have real time requirements and you can't use a traditional `malloc` implementation. But trust me, you're going to hate the day you've heard about those pools. Pools have arbitrary size limits (at most 20 objects of this, at most 30 objects of that...). Everybody is going to have a paranoia attack and set the limits to huge values, and then you're out of memory, and then you start thinking about the "right" limits, and it turns out that it's extremely hard to figure that out, because seemingly independent modules have related memory requirements (say, they handle mutually exclusive situations so you'd like them to use the same memory, but how?), and then you need some hairy logic to compute those values on a per-configuration basis, which means building several versions of the program, and maybe creating scripts for computing the values at build time, and you're going to *hate the day you've heard about those pools*.

If you're absolutely sure you can't create a single allocator managing your memory, at least *don't allocate objects of classes with constructors* from pools, use C-like structs instead. If you refuse to give up and admit that you're doing a pretty low-level thing and instead want to keep pretending that you're "programming in the problem domain", and for some reason you think C++ classes help you with that - go ahead. The punishments for your crime are discussed throughout the FQA.

Assignment operators

This section is about `operator=`. It's rather strange that of all possible things, people only "frequently ask" about self assignment, but those are the only questions listed in the FAQ.

[12.1] What is "self assignment"?

FAQ: It's assigning the object to itself, directly or indirectly. For example:

```
void f(C& x, C& y) { x=y; }
void g(C& x) { f(x,x); }
```

FQA: The tricky part is that the assignment operator may be overloaded, and you are expected to overload it in the classes that "own" resources (most frequently memory obtained with `new`). Incidentally, these are exactly the cases when self assignment may lead to trouble, which is why we have the next question. Which will also show that the FAQ's definition of "self assignment" is unfortunately too conservative, as "self" is a vague concept.

[12.2] Why should I worry about "self assignment"?

FAQ: If you don't, your `operator=` will probably misbehave severely upon self-assignment:

```
YetAnotherSoCalledSmartPointer& YetAnotherSoCalledSmartPointer::operator=(const YetAnotherSoCalledSmartPointer& p)
{
  delete _p;
  _p = new TheThingPointedByTheSmartPointer(*p._p);
  return *this;
}
```

Self assignment will first delete `_p`, and then dereference it, which is nasty. And it is all *your* fault - you should have handled self assignment!

FQA: Oh, so it's my fault, isn't it? Then why isn't this question listed in any other programming language FAQ on the planet? I mean, it's the same me all the time, it's the languages that are different. So why don't we lose the feelings of guilt and check what it is about C++ that causes the problem?

Most languages daring to call themselves "object-oriented" have garbage collection, so you don't spend your time writing stupid functions for deallocating stuff, then copying stuff. These functions are all alike, but with C++ you get to write this kind of code over and over again.

C++ doesn't have garbage collection, so a problem arises - if people are creating all those objects, how are they going to dispose them, especially if they overload operators and do things like `a+b+c`, without even keeping the pointers to the temporary objects created when sub-expressions are evaluated? The C++ answer is to have an "owner" object for each chunk of allocated memory, and let `operator=` and the destructor worry about freeing the memory. The most common manifestation of this quite unique language design decision is unnecessary copying, which happens when objects are passed to functions or returned by value or "cut out" from larger data structures.

The self assignment problem is another manifestation of this same thing. If objects were passed by pointers/references and garbage-collected automatically, self assignment (setting the pointer to the value of itself) would be harmless. And the worst part is that there are actually more cases of self assignment than `x=x`. For example, consider `std::vector::push_back()`. What if someone passes an object from the vector itself, as in `v.push_back(v.back())`? `push_back` may need to allocate more memory, which may cause it to free the buffer it already uses, destroying its argument. Where does the "self" of an object end?

You have to be really smart to get all those smart pointer and container classes right. Too bad it's still worse than good containers built into a language.

[12.3] OK, OK, already; I'll handle self-assignment. How do I do it?

FAQ: By adding a test like `if(this==&that) { return *this; }` or by copying the members of the input object before deallocating members of `this`.

FQA: This avoids the destruction of your function argument via self-destruction in `operator=`. However, there are other cases when that can happen - consider `std::vector::push_back()`.

When performance is more important to you than simplicity & generality, one way to deal with this family of problems is to document them and pass them on to the user of your code ("it is illegal to push an element of a vector into that vector, if you need to do it, copy the element explicitly").

When simplicity & generality are more important than performance, you can use a language automatically tracking references to objects, guaranteeing that used objects won't get destroyed.

Operator overloading

This section is about operator overloading - a way to make the code "readable" as long as the reader doesn't care what the code actually does.

[13.1] What's the deal with `operator` overloading?

FAQ: Overloaded operators provide an "intuitive interface" for the users of your class. They also allow templates to work "equally well" with classes and built-in types.

The idea is to call functions using the syntax of C++ operators. Such functions can be defined to accept parameters of user-defined types, giving the operators user-defined meaning. For example:

```
Matrix add(const Matrix& x, const Matrix& y);
Matrix operator+(const Matrix& x, const Matrix& y);
Matrix use_add(const Matrix& a, const Matrix& b, const Matrix& c)
{
  return add(a,add(b,c));
}
Matrix use_plus(const Matrix& a, const Matrix& b, const Matrix& c)
{
  return a + b + c;
}
```

FQA: Operator overloading provides strong source code encryption (the time needed to figure out what `a+b` actually means is an exponential function of the number of types, implicit conversions, template specializations and overloaded operator versions involved).

It is also one way to have templates malfunction equally miserably with user-defined and built-in types - if `BraindeadPseudoTypeSafeIterator<T>` supports the syntax `*p` and `++p`, it can be used just like the built-in `T*` in a template "algorithm" like `std::for_each`. However, it could also work without operator overloading - for example, you can have a global `increment` function for all "iterator" types, and implement it for all pointer types using a global template wrapper calling `++p`. Which means that operator overloading is pure syntactic sugar even if you don't consider templates a pile of toxic waste you'd rather live without.

This syntactic sugar can only be added to a language like C++ together with more than a grain of salt. For example, what if `a+b` fails? There's no good way to return an error status, because the language is "strongly typed", so `a+b` always returns objects of the same type, for example, a `Matrix`. There's no natural "bad matrix value", and anyway the whole point of `a+b` is that you can also write `a+b+c`. If you need to test each addition with an `if` statement, `a+b` is not much better than `add(a,b)`. What about throwing an exception upon error? Not a bad idea, in a language *supporting* exceptions, as opposed to merely providing non-local `goto` using `throw/try/catch` syntax and having *you* care about "exception safety" in each and every bit of code.

What about the allocation of our result - where does that `Matrix` live? If it's allocated dynamically with `new` and returned by reference, who is supposed to clean it up, in particular, who keeps the list of all temporaries created by `a+b+c...`? But if it's returned by value, then the copy constructor is going to copy lots of matrices. Bad if you care about performance. And if you don't, you *surely* wouldn't mind the smaller run time overhead of *garbage collection* (not to mention run time boundary checking and other kinds of safety belt), so you'd probably choose a different language in the first place.

Operator overloading is not necessarily a bad idea - if you can actually keep the promise about "readability". To do that, you need at least three things: comprehensible overload resolution rules, easy-to-use exceptions and easy-to-use automatic memory management. C++ offers none of those.

[13.2] What are the benefits of operator overloading?

FAQ: You can "exploit the intuition" of the users of your classes. Their code will speak in the language of the problem instead of the language of the machine. They'll learn faster and make less errors.

FQA: The keyword in the FAQ's answer is "exploit". Your intuition tells you that `a+b` just works - and in the "problem domain", you are right. But in C++, it doesn't take the machine too long to raise its ugly iron head and start talking to you in its barbaric language, and it has all the means to make you listen.

If you want to program "in the problem domain", use a language designed for that problem domain. Prototype numerical algorithms in Matlab or the like, not in C++ with some matrix library. Design hardware in Verilog or the like, not SystemC (which is C++ with some hardware description primitives library).

The next best alternative is to use a general-purpose language where operator overloading and other metaprogramming features actually work. It is typically worse, but may be cheaper - special-purpose tools are used by less people than general-purpose ones, so the vendor must charge each user a higher price. And it will make you happy if you're one of those people who think that there's nothing a special-purpose language can do that can't be done equally well or better using the wonderful metaprogramming facilities of the awesome language X (typically *wrong*, but strong programmers can be very productive operating under this assumption - unless X=C++).

Operator overloading and other features sure make C++ equally adaptable to any problem domain. This is achieved by making it the wrong tool for every job.

[13.3] What are some examples of operator overloading?

FAQ: Here are a few (some real, some hypothetical), there are many more.

- `str1 + str2` concatenates a couple of `std::string` objects.
- `NapoleonsBirthday++` increments an object of class `Date`.
- `a*b` multiplies two `Number`s.
- `a[i]` accesses the i'th element of a user-defined `Array` class object.
- `x = *p` dereferences a "smart pointer" which actually represents an address of a disk record; the dereferencing seeks to that location on the disk and fetches the record into `x`.

FQA: C++ operator overloading is a bad way to implement all of these things. It's an equally bad way to implement almost everything that comes to mind.

- *Strings*: a general purpose high-level language should have a good built-in string type, because text processing is a very common task. In particular, a string is an extremely common type of function parameter, so if you have no built-in string type, each kind of interface will use its own kind of string, and the users will have to spend most of their time converting between string types. Even `char*` and `std::string`, which are both *standard* string types (there are *lots and lots* of non-standard ones), have interactions that operator overloading fails to hide. For example, `dir + "/" + file` compiles with `std::string dir` and `char* file`, but fails to compile with `char* dir` and `std::string file`, "exploiting" the intuition of users.
- *Dates*: what does `date++` *mean*? Does it add a day or a minute or a second? This question is especially interesting if someone had a severe modeling-the-universe-using-type-systems attack and defined a `Time` class (incremented by seconds), a `Date` class (incremented by days) and implicit conversions between them. And if you know that dates are

incremented in day intervals, so 1 means "one day", why not just use `int` instead of `Date`? "Encapsulation" - of what exactly?
- *Numbers*: should be built into your language, or you should stop pretending that you're "programming in the problem domain" (both approaches are perfectly legitimate). See the discussion about strings.
- *Arrays*: should be built into the language, too. And if you use a tricky data structure because resizable arrays built into your language are not good enough, making it look like an array is not necessarily a good idea. For example, people might want to find the definition of the `operator[]` in `a[i]`. What should they do - search for "["? Oh, they are using an IDE that actually understands 75% of C++ syntax? Now what - select the opening bracket and ask for its definition? Never worked for me.
- *Smart pointers to disk records*: I wish all C++ weenies used an operating system which implements file systems using this advanced technique. This way, whenever the "dereferencing" would fail, either the error wouldn't be reported, or an exception would be mishandled, and the weenies would lose their files, which would be good for everybody.

Basically most operator overloading use cases fall into one or more of the following categories:

- the thing can't be implemented well unless it's built into the language (arrays, strings, numbers, "smart pointers" doing things like reference counting)
- the thing is tricky and shouldn't be confused with built-in types, so the interface should be different ("smart pointers" doing things like disk access, complicated data structures)
- the thing is obscure and shouldn't be implemented at all (incrementing dates by unclear amounts of time, transferring output streams into hexadecimal mode without an option to restore the previous mode)

However, it doesn't mean that operator overloading is never useful - that is, in the languages that can actually support it.

[13.4] But `operator` overloading makes my class look ugly; isn't it supposed to make my code clearer?

FAQ: No - it's supposed to make the code *using* your class clearer! For example, you may claim that `T& operator[](int i)` is less readable than `T& getAt(int i)`, but surely `arr[i]` is more readable than `arr.getAt(i)`!

You should realize that "in a reuse-oriented world", *usually* (yes, the FAQ says "usually") many people will use your class, but only one writes its code (that one person is you). Think about the good of the majority - your users!

FQA: Those guys living in the "reuse-oriented" worlds are very noble characters. Too bad we're stuck on planet Earth, which didn't seem to have any particular "orientation" last time I checked. On this planet, most classes you write are used exclusively by yourself, the majority of the rest is used by two or three people, and once in a while you get to write a class to be used by N people for N>3 - typically it's not an array class, but something with *a little bit less trivial functionality*. Here on Earth, this is considered *good*: the more interaction between people and components, the more errors there are likely to be (insert the FAQ's favorite pseudo-business-oriented statements about "defect rate" here).

Of course this doesn't mean that *the other* claims make any sense. For example, users have to figure out what your class does before they use it, so its interface must be defined in a readable way, not just look readable when used. And your users will probably need to understand your implementation, too, because even when your code no longer has bugs (this might take a while), bugs in other pieces of code may end up corrupting *your* data (this is C++), so people will have to inspect what's left of that data in order to find the original error. So an unreadable and/or complicated implementation is not a very good idea even in a "reuse-oriented" world.

Last but not least, `arr.getAt(i)` is *not that bad*.

[13.5] What operators can/cannot be overloaded?

FAQ: Most can be overloaded. You can't overload the C operators `.` and `?:` (and `sizeof`). And you can't overload the C++ operators `::` and `.*`. You can overload everything else.

FQA: There are other restrictions, for example there can't be a global `operator[]` overload - you can only overload this operator using a class member function. Which is not necessarily bad.

[13.6] Can I overload `operator==` so it lets me compare two `char[]` using a string comparison?

FAQ: No, at least one parameter of an overloaded operator must be a user-defined type.

But even if you could do it, you shouldn't - you should use a class like `std::string`. Arrays are evil!

FQA: C++ doesn't let you do it for a good reason - if you could do it, how would you make sure that your program is compiled consistently in the sense that your operator is called *everywhere* - in all contexts calling `operator==` on `char[]` arguments? And what if two such operators are defined by two different modules?

Unfortunately, these problems are almost equally severe with user-defined types. For example, `std::map` doesn't have an overloaded `operator<<` printing it to an `ostream`. You can define one, but so can I, and then our libraries are linked together into a single program. The result is unpredictable, ranging from a link-time error to a program printing some maps using your code and some with mine.

As to the "arrays are evil" mantras - how about chanting them to the authors of the C++ standard library that defined an `std::ifstream` constructor accepting `const char*`, but not `std::string`? And they didn't define an `operator const char*` in `std::string`, either. `std::ifstream in((std::string(dir) + "/" + file).c_str());`. Give me a break.

And why does the FAQ say "a `std::string`-like class", not just "`std::string`"? Because if it did, the majority of the audience stuck with one (or more) of the many, many "string-like" classes developed before one was added to the C++ standard library would laugh quite bitterly.

[13.7] Can I create a `operator**` for "to-the-power-of" operations?

FAQ: You can't. You can only overload operators already in C++, without changing the number of arguments, the associativity or the precedence. If you think it's wrong, consider the problem of `a**p`, which means "a times the result of dereferencing p". If you could define `operator**`, you'd make such expressions ambiguous.

And you know what? Operator overloading is just syntactic sugar. You can always define a function instead. Why not overload `pow`?

Or you could overload `operator^`, which works but has the wrong precedence and associativity.

FQA: Sure, there's lots of grammar in C++, and making an unambiguous addition is almost impossible. The fact is that, informally speaking, there's a finite amount of "good" syntax you can have in your language. This is a valid argument if you advocate domain-specific languages (which can make the short, simple syntax map to stuff most useful in that domain), or if you argue that "languages should have no syntax" (Lisp, Smalltalk and Forth are examples of languages with almost no syntax). However, it's funny when this argument is used by the C++ FAQ. It is hardly compatible with the claims that C++ is applicable everywhere. And then there's the *huge* amount of *completely pointless* complications in the C++ grammar (like the constructor/function declaration puzzle).

What's that? "Operator overloading doesn't provide anything fundamental, it's just syntactic sugar"? Now you're talking. Make that "syntactic cyanide" and I'm in.

Overload `pow`? Can somebody tell what's the *point* of all this overloading? If you want obfuscation, there are tools you can use that allow you to keep the source in the unobfuscated form, which may be very handy at times. Besides, isn't it `std::pow`? AFAIK you are not allowed to add names to the `std` namespace. And if you add a global function pow, you'll lose the precious "transparency" (`std::pow(x,y)` will never call your version of `pow`, only `pow(x,y)` will), so what's the point?

The idea to overload "bitwise exclusive or" to mean "power" is just stupid. I wonder where they get these ideas. It's as if someone decided to overload "bitwise left shift" to mean "print to file". Wait a minute - they did that, too... Oh well.

[13.8] Okay, that tells me the operators I *can* override; which operators *should* I override?

FAQ: You want to help your users, not confuse them. Overload operators if it makes the life of your users easier.

FQA: Translation: don't overload C++ operators. While we're at it, don't overload C++ functions, either. C++ overload resolution *always* ends up confusing users. And you can't add error handling without using the horrible C++ exceptions, and you can't allocate objects simply and efficiently.

[13.9] What are some guidelines / "rules of thumb" for overloading operators?

FAQ: 22 (!!) "rules of thumb" are listed (actually 20, 2 just say "use common sense"). Basically, the rules are about making overloaded operators behave similarly to built-in operators - if you define +, make it associative and don't change the parameters, etc. The FAQ warns that the list is not exhaustive, nor should it be interpreted as strict rules that can't have exceptions.

FQA: Rule of thumb #23: things which have simple functionality but are very hard to get right *linguistically* should be in the compiler.

Use common sense. Do you have the time for all that thinking when all you get is "syntactic sugar" of questionable taste? How about implementing some user-visible functionality instead? Not only is it what people (customers, employers, colleagues, friends) ultimately care about, it's also much more fun.

[13.10] How do I create a subscript `operator` for a `Matrix` class?

FAQ: Use `operator()` which can get two indexes, i and j. Don't use `operator[]`, which can only get one index. A code listing follows.

FQA: Oh, dear. You're writing a `Matrix` class. My condolences.

Once you define your subscript operator, be prepared to answer many more questions. How do you allocate the result of `operator+`? How do you map expressions like `A+B*C` to optimized implementations of several operations (for example, `multiply_add`)? Why are you writing a `Matrix` class with overloaded operators instead of prototyping the code in Matlab or the like and then implementing the prototype in C so that it runs fast and anybody can tell how and why from the code, instead of figuring out how `A+B*C` actually works?

[13.11] Why shouldn't my `Matrix` class's interface look like an array-of-array?

FAQ: The "array-of-array" alternative uses `operator[]` to return an array object, which in turn has an `operator[]` returning a single element.

This approach is likely to be worse because of performance issues having to do with the optimal layout of the matrix in memory, says the FAQ. Or maybe it says something else similar to it. It talks a lot about the issue.

FQA: You can't implement the array-of-array syntax with a single function call, because C++ has no `operator[][]`. You need to return some temporary object with an `operator[]`.

This means extra work for you, because instead of writing a function you write a whole class and a function returning its objects. This also means extra work for the user who has to read all these interfaces. This also means extra work for the compiler. The chances of a compiler from planet Earth to optimize out the temporary objects are very close to zero. Which means it's also extra work for the machine running your code.

And what did you achieve? "Syntactic sugar". This reasoning can frequently be used to optimize the entire `Matrix` class with all its overloaded operators out of your schedule. And it doesn't depend on performance-related claims about memory layout and stuff.

[13.12] I still don't get it. Why shouldn't my `Matrix` class's interface look like an array-of-array?

FAQ: Because it makes checking the arguments or changing the data structure harder. The FAQ then explains how this can still be done, and claims that the compiler will optimize the temporaries out, but doesn't like the fact that it's a lot of work.

FQA: The FQA primarily doesn't like the fact that it's a lot of work. But it would also like to point out that: 1. The compiler will probably *not* optimize the temporaries out. 2. Key performance-critical data structures are extremely costly to change no matter how much "encapsulation" you use, because optimizations depend on the representation. 3. Checking the arguments is *always* hard with operator overloading, because what would `operator()` do once an error *was* found - return -1 (how?), set a global error flag, throw a C++ exception? All these options are not really acceptable.

Anyway, if you want to waste some time without getting useful work done, there are ways to do it that are much more fun than getting `operator[][]` sort of work. And this is where the FAQ and the FQA agree.

[13.13] Should I design my classes from the outside (interfaces first) or from the inside (data first)?

FAQ: Outside, of course.

A long discussion about the accessors one should have in a `LinkedList` class follows. A ground-breaking conclusion is reached: you should give accessors to the elements of the list, but not the "infrastructure" (say, the pointer to the first node).

FQA: *What?* What does this question have to do with operator overloading? And what does the kind of accessors you want to have in your class (the final code) have to do with the way you design it (think before you code)?

There's no single good way to design classes or any kind of software. Sometimes the interfaces are the important thing, sometimes the implementation, and sometimes both. Designing is a kind of thinking. Sometimes you need to think about the data in order to figure out what kind of interfaces are ultimately possible to implement efficiently. Sometimes you must figure out the interface first to make sure the whole thing is actually useful. Frequently you have to think about both and possibly iteratively refine them.

The idea that implementations are not important (because you can always change them or for other reasons) may only emerge from implementing boringly trivial things, or from doing nothing (one way of doing nothing in the software industry is to obsessively seek the best way to spell the iteration over the elements of a list for a living).

By the way, the FAQ mentions "a billion-line app". C++ doesn't scale to many lines of code in the sense that a large monolithic C++ application (one where all code runs in a single address space) can hardly be maintainable. This claim may be made about other languages as well, but the problem is especially severe with languages assuming unmanaged environments, where a buffer overflow in module A may silently corrupt the data of module B. If you have lots of unmanaged code, breaking it into processes can save several tons of your bacon (of course you have to build the system from the beginning that way - retro-fitting it is extremely expensive).

[13.14] How can I overload the prefix and postfix forms of operators `++` and `--`?

FAQ: Like this:

```
Iter& operator++ ();     // ++p
Iter  operator++ (int); // p++
```

The postfix version should return something equivalent to the copy of `*this` made before the operator call.

FQA: Silly syntax, isn't it? If you do overload operators, please read and follow all the boring rules about their syntax and semantics. This will help you soften the blow on your users, or even fall asleep and forget about the whole thing if we are lucky.

[13.15] Which is more efficient: `i++` or `++i`?

FAQ: It's the same for built-in types. `++i` may be faster for user-defined types, because there's no need to create a copy that the compiler may fail to optimize out. Most likely the overhead is small, but why not pick the habit of using `++i`?

FQA: Oh, suddenly the all-mighty compiler can't optimize out a temporary! I actually picked that habit at some point. That was before I picked the even more useful habit of avoiding overloaded C++ operators.

The really interesting questions are about the performance of `A+B*C` compared to `D=B;D*=C;D+=A` compared to an optimized `multiply_and_add_matrices(A,B,C)` function.

Friends

The questions here are about `friend` declarations. A short answer to them all: you probably don't need `friend` declarations.

[14.1] What is a `friend`?

FAQ: A class can declare other classes and/or functions as friends. The declaration allows the friends to access non-`public` class members unaccessible to other code outside of the class.

FQA: In other words, `friend` refines the access control provided by the `private` keyword. Which means it's almost useless, simply because `private` is almost useless. Specifically, it fails to provide *encapsulation*, and therefore is little more than a *comment*.

If you define interfaces between modules developed independently and want these interfaces to be stable, it's better to use C for the interface definitions, not C++. If you insist on using C++ (forcing both modules to be built with the same tools), relying on `private` to provide encapsulation is a bad idea, because changing private members triggers recompilation of the client code. You can use forward declarations and/or pure abstract classes to provide a more stable interface.

If you work on the internal interfaces between the different classes of a module, and you're stuck with C++, using `private` is surely better than defining all members `public`. This way, someone can easily tell that a member is not accessed outside of the class, and clarity is good. However, there is little reason to obsess with the fine details of access control in the internal interfaces.

Most of the time, the distinction between `private` and `public` is good enough to describe what you want. If you feel a strong urge to split tightly coupled functions between several classes, you can do it with `friend`, or you can make the members `public`. Of course it's possible to argue forever about this ("But it's bad design to allow *everyone* to access the members!", "But it's bad design to have tightly coupled classes!"). However, these are some of the internal classes of a module, nothing more, nothing less. If the module has reasonable size, the access control is not that big a deal. If the module is huge, you have a huge problem either way, especially with C++, which compiles forever and doesn't localize the damage of run time errors.

Therefore, `friend` is totally useless for the external interfaces and almost useless for the internal interfaces. So there you are.

[14.2] Do friends violate encapsulation?

FAQ: On the contrary - they enhance it. If used properly, of course.

For example, you can use them to split code into two classes and have their data inaccessible for code in all other classes. Or you can use them as "a syntactic variant" of member functions.

Think of the friends as part of the class interface instead of something outside the class.

FQA: What encapsulation? C++ doesn't have encapsulation. It has access control (code outside of a class using `private` members of this class will not compile), but it has neither compile time encapsulation (stable binary interfaces) nor run time encapsulation (safe memory access). What's there to violate?

The case of two classes was discussed in the previous FAQ; this argument only sounds convincing if you think C++ classes are a good way to define the important, stable interfaces in your system.

As to the "syntactic variant" business, it's probably about overloaded operators. The C++ syntax makes it impossible to define overloaded operators as class members unless the first argument is an object of the class. Since C++ uses `stream<<obj` to print things, many classes declare a `friend operator<<`, for example. This argument is only interesting if you think C++ operator overloading is any good.

[14.3] What are some advantages/disadvantages of using `friend` functions?

FAQ: Advantage: they allow the designer to choose between `obj.func()` and `func(obj)`, which "lowers maintenance costs".

Disadvantage: they require more code to achieve dynamic binding. Non-member functions can't be `virtual`, so if the designer's syntax of choice is `func(obj)`, and dynamic binding is needed, `func` will have to delegate to a `protected virtual` member with `obj.func()`.

FQA: Listen carefully, this is an excellent opportunity to learn about the C++ approach to writing software. Choosing between two equivalent syntactic forms is called "design". The availability of many forms and the "right" choice between them is expected to "lower maintenance costs". Does this make any sense at all? Have you seen headlines such as "An engineer replaced `obj.func()` with `func(obj)`, profits sky-rocket" or "A researcher wins the Turing award for discovering a new way to replace `obj.func()` with `func(obj)`"?

And why not say out loud that the real choice is between `obj<<stream` and `stream<<obj` or something like this? This is really about operator overloading, because that's where the member/global function distinction matters most syntactically. Why not just say `obj.print(stream)` instead of shifting streams by the amount of bits in an object or something?

[14.4] What does it mean that "friendship isn't inherited, transitive, or reciprocal"?

FAQ: It means that classes derived from a friend class don't automatically become friends (do *you* trust the kids of your friends?), a friend of a friend doesn't automatically become a friend (do *you* trust the friends of your friends?), and that a class declaring another class as "friend" doesn't automatically become a friend of that class (do *you* trust anyone who calls you a friend?).

FQA: The real life analogies are too scary for a detailed discussion. "Do *your* friends have access to your members"?

One reason to avoid friends is that the only possible benefit is a little extra clarity, but this benefit can be dwarfed by the extra obfuscation - not everyone is (or should be) 100% sure what `friend` means when it comes to class hierarchies and stuff.

[14.5] Should my class declare a member function or a `friend` function?

FAQ: Use a member function unless you must use a `friend` function. For example, you may need `friend` for operator overloading.

FQA: The FAQ says it at last! See? It was about operator overloading all the way.

The FAQ's advice may be further simplified if we use the observation that C++ operator overloading is just a pain in the neck.

Input/output via <iostream> and <cstdio>

This section explains the benefits of `iostream` - a *"type-safe"* I/O library (which does not mean it will *save* keystrokes when you *type* some code).

[15.1] Why should I use `<iostream>` instead of the traditional `<cstdio>`?

FAQ: There are four reasons:

- *Increase type safety*: with `iostream`, the compiler knows the types of the things you print. `stdio` only figures them out at run time from the format string.
- *Reduce the number of errors*: with `stdio`, the types of objects you pass must be consistent with the format string; `iostream` removes this redundancy - there is no format string, so you can't make these errors.
- *Printing objects of user-defined types*: with `iostream`, you can overload the operators << and >> to support new types, and the old code won't break. `stdio` won't let you extend the format string syntax, and there seems to be no way to support this kind of thing in a way avoiding conflicts between different extensions.
- *Printing to streams of user-defined types*: you can implement your own stream classes by deriving from the base classes provided by `iostream`. `FILE*` can not be extended because "it's not a real class".

FQA: Why should I do this, why should I do that, you ask. What kind of manners are these? Do what you are told.

Assignment Number 1 - convert all your evil `printf("0x%08xn", x)` statements to this:

```
std::cout << std::hex << std::setfill('0') << std::setw(8) << x << std::dec << std::endl;
```

Even if you commit the environmental crime of namespace pollution, adding a `using namespace std` and removing those pesky `std::`, the verbosity is still amazing. This achievement is not accidental. It follows from one of the basic flaws in the C++ way of thinking: the "everything is a type" axiom. For example, `hex` has a special type which hexes streams, and so does every other strange object sent to `cout`.

The FAQ explains why this thinking is good for you. Here's why the FAQ is wrong:

- Why is *type safety* a good thing here - because we gain performance? The last time I checked, we didn't really gain any - your typical iostream implementation is slower and bulkier than your typical stdio implementation. Anyway, isn't I/O the bottleneck here?
- Oh, now I get it - type safety is supposed to help the compiler *catch errors*. This is very important for people who never actually look at what they print. The needs of this population surely justify the 700K of crud that `gcc` gets to parse when compiling a C++ "Hello, world" problem, as well as my time spent waiting for it to compile. What's easier to fix: a run-time printf problem or a compile-time iostream problem (with the compiler helpfully listing hundreds of overloaded left shift operators)?
- Maybe "everything is a type" is a good thing because it lets you *print objects of user-defined types*. If you believe this, concentrate for 10 seconds - this is your chance to achieve enlightenment. **What on Earth prevents you from printing user-defined objects with printf?** You write a helper function which gets a pointer to your object and prints it. While you're at it, give the function some special name, so that you don't have to shovel through lists of overloads anymore. What's that? It's verbose, because function calls are separate statements, unlike the funky left shift operators? Well, look at a simple statement printing an integer above if you care about verbosity.
- What about *printing to streams of user-defined types*? Well, this claim is valid in the sense that there's no way to define custom `FILE*`, but it has nothing to do with the fact that "it's not a real class". A "real class" is not a very well-defined term; to me, C++ classes look pretty surreal compared to classes defined in almost any other language I've ever seen. Anyway, stdio could provide ways to define custom `FILE*`, for instance by providing `read`, `write` and other callbacks. The authors just didn't bother. The ability to define custom streams does not justify the problems of iostream, nor is it a feature unique to iostream. This ability is achieved in (relatively) sane OO ways, (almost) in the way it's done in many languages saner than C++ and I/O libraries saner than iostream.

`iostream`. The only thing you'll gain from all this extra typing is extra long build cycles and error messages and extra large program image. This is what you get when you shift a file object by an integer.

[15.2] Why does my program go into an infinite loop when someone enters an invalid input character?

FAQ: Probably you did something like `int i; std::cin >> i;` without checking for an error. From now on, all attempts to read stuff from `std::cin` won't actually read anything, because the stream has entered an erroneous state. Presumably, you wrote a program that will still keep trying. Use something like `while(std::cin >> i)` instead.

FQA: Yep, that's iostream's way to handle input errors. For some reason, it doesn't throw an exception (it's not really bad, because what's really bad is C++ exceptions). But it doesn't return an error code, either, because the overloaded operator has to return its first argument for the "neat" operator call chaining to work. So it has to set an error bit - the dreaded "zombie object" pattern the FAQ condemns so convincingly.

You can then check for errors using `if(std::cin)` or the like because the object has overloaded `operator void*`, and `void*` is convertible to `bool`. You can also check for end-of-file conditions this way. Actually you can check for error-or-end-of-file conditions. The more operator calls you chain, the less details about the error context are left. Simple, safe, efficient and elegant.

Oh, and iostream has other state bits. For example, `std::hex` transfers it to hex mode. People frequently forget to undo the effect, which is especially entertaining when they print numbers like `10`, which is as good a decimal ten as it is a hexadecimal sixteen. Well, if you found the bug (which for some reason went undetected by the type-safe almighty paranoid C++ compiler), you can use `std::dec` to get the stream back to decimal mode. Um, actually, it transfers it to decimal mode, which is not necessarily "back", because we don't know which mode it was in before it was hexed. This gets interesting when people print numbers and user-defined objects in the same statement, and count on their hexing to maintain its effect after the user-defined object is printed.

[15.3] How can I get `std::cin` to skip invalid input characters?

FAQ: Here's an example:

```
    while ((std::cout << "Give me your credit card number now!!") && !(std::cin >> n)) {
      std::cout << "Don't mess with me, I'm written in C++!!!";
      std::cin.clear();
      std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
    }
```

FQA: Oops, you forgot to test for end of file. If a brave user hits `Ctrl-D` or whatever it takes to close the standard input with the given terminal, your program will enter an infinite loop. Overloaded operators and error handling are not very compatible.

`std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');` speaks for itself. You just can't invent it if your "use case" is yourself getting any work done.

[15.4] How does that funky `while (std::cin >> foo)` syntax work?

FAQ: `istream` has overloaded `operator void*`. The compiler calls this operator in boolean contexts (when it expects a condition, for example), because `void*` can be converted to a boolean. The operator returns `NULL` when there's nothing left to read, or when a format error occurred previously.

FQA: There's nothing "funky" about it. It is ugly and boring. It's also scary because many people think this is what programming is all about - using complicated syntax to do simple things without even getting them right (how do you tell end-of-file conditions from format errors?).

Why is it `operator void*`, and not `operator bool`? Apparently because the compiler implicitly converts booleans to numbers in "numeric contexts" (such as `file1+file2`), and we don't want that to compile, do we?

But wait, there's more! There's an actual book out there, called "Imperfect C++", arguing that `operator void*` is not the way to go, either. Because this way, `delete file` would compile. Surely we don't want it to, do we? I mean, the fact that the statement is completely *moronic* shouldn't matter. Morones have a right to get an equal opportunity in the exciting world of C++ programming; let's catch of all their errors at compile time. Evil people spread rumors about the problem being undecidable, but we should keep trying.

[15.5] Why does my input seem to process past the end of file?

FAQ: Because the stream doesn't know that you've reached the end of file until you actually try to read past the end of file. For instance, a stream object may read characters which are entered interactively using a keyboard. So it's impossible to tell when it's over.

FQA: That's right, and has nothing to do with iostream - except the fact that iostream handles I/O errors in the silly way discussed above.

[15.6] Why is my program ignoring my input request after the first iteration?

FAQ: Because non-digits are left in the input buffer. You can ignore them using calls like `std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');`

FQA: Doesn't the fact that there are so many questions about the handling of input format problems ring a bell? This silent treatment of errors is wrong. Stuff like parsing is better handled by languages which have good support for exceptions (built-in strings and containers help, too). If you have to live without exceptions, at least the library should treat errors reasonably. The iostream interface of operator call chains makes error handling as awkward as it gets.

[15.7] Should I end my output lines with `std::endl` or `'\n'`?

FAQ: The former has the additional side-effect of flushing the output buffer. Therefore, the latter will probably work faster.

FQA: The latter is also shorter, unless you have a `using namespace std`. Many people probably ask this question because they think that `endl` will end the line the way it's normally done at a given platform (say, emit CRLF on Windows and LF on Unix). When `endl` behaves that way, so does `'\n'`. You can suppress this behavior when you open the file (pass `ios::binary` with iostream or the `"b"` flag with `fopen`).

[15.8] How can I provide printing for my `class Fred`?

FAQ: By adding a `friend std::ostream& operator<< (std::ostream& o, const Fred& fred);`. The operator can't be a member of the class, because we need the object to be the second argument, not the first one.

FQA: Yeah, yeah, C++ and its stupid syntax. Let's forget about it for a moment: the real problem here is semantical, and it starts at the question "How can I provide printing *for my class*?". The formulation encourages to give an answer in the spirit of "everything is a type", and indeed this is the kind of answer given by the FAQ. Here's *the* way to print *all* objects of your class.

But was that really your question? What if you want more than a single output format for your class? For instance, you might want a textual format for people and a binary format for machines. Or there could be several textual formats - even integers can be printed using different bases. Do you really want to develop and maintain the layers of syntax needed to make things like `hex` and `dec` sort of work? Why not just define named functions for printing?

The `friend` thing is also questionable. With all due respect, you might want to print to several different output channels, *without* creating an adapter class derived from `ostream`. One simple reason for such heretical behavior is performance - `ostream` may be too slow for things like real-time logging. Why not create accessors to the data you want to print, so that all the different output methods won't need to be part of your class, which may then have a stable interface? Encapsulation? You've already made your data public - one can read it by printing to a `stringstream`. Providing accessors violates obfuscation, not encapsulation.

If you abandon the `friend` part and the `operator` part, the remaining part of the FAQ's answer - create a global function accepting some kind of stream object and an object of your class and do the printing there - makes a good advice. In particular, it may be better than a method for the same reasons making it better than a friend (however, unlike a friend, a method may be polymorphic, which might be useful).

[15.9] But shouldn't I always use a `printOn()` method rather than a `friend` function?

FAQ: Not "always". A method may be useful for a class hierarchy, for example, but in that case it would normally be protected, not public.

Some people think `friend` violates encapsulation, and therefore prefer to have their `operator<<` call a public method of the class, even though there's no class hierarchy. Their basic assumption is wrong, so they end up writing extra code that doesn't solve any real problems and confuses people.

FQA: Just to make things clear: a `printOn()` method is something like `void Fred::printOn(std::ostream& o) const`.

The FAQ is right about the important thing: creating extra layers of code doing nothing but delegating work to other layers of code is a pesky habit. Many people feel that "good design" means "lots of code" or "lots of layers". Actually, those are some of the meanings of "bad design".

The fact that `friend` is really needed here to compensate for syntactical limitations of C++, as well as the problems with the whole `friend operator` approach, are discussed in the previous FAQ. They are not the main theme here. The main theme here is the message to the "designers": listen to the voice of C++ overlords who originally inspired you to create N layers with 0 functionality, and stop.

[15.10] How can I provide input for my `class Fred`?

FAQ: By adding a `friend std::istream& operator>> (std::istream& i, Fred& fred);`

FQA: This defective technique is symmetrical to the `operator<<` thing, which was discussed above. We'll use this opportunity to look at some advanced defects of this approach. For beginner's defects, go two FAQs back.

The complicated problems come from the same source as the simple problems: the idea that for each type, we want exactly one output function looking like a bit shift operator. Therefore, to create complicated problems, we'll need some complicated types. Luckily, C++ comes with one of the most complicated type systems in the solar system (itself an example of a simpler system).

Consider `std::map`. It's standard, and so is iostream. Does this mean that the operators reading and writing them are also standard? I think you know the answer. But no matter: you can create a template operator printing all kinds of `map`. So can I. We're both happy, unless we have to integrate our code into a single program some time later. We both used the same intuitive name - `operator<<` - for our printing functions. Normally we'd rename one of the two, but in this special case, this is *really* going to be ugly - for instance, other templates will break because they think everything can be printed with `<<`, and even if there aren't any, `printMap(std::cout << "map: ", myMap) << std::endl` is too ugly even for seasoned C++ developers, who do have pretty low standards.

The real fun starts when two people overload (or specialize or whatever they call it) the operators for *special cases of types* in conflicting ways. For instance, I know how to print `std::map<std::string,T>`, and you know how to print `std::map<T,int>`.

The question is, who gets to print `std::map<std::string,int>`? The compiler is in trouble, and when a C++ compiler is in trouble, it immediately empties its bowl, dumping a nice, large, stinky error message right at our faces. Happy, happy, joy, joy, dear colleague.

[15.11] How can I provide printing for an entire hierarchy of classes?

FAQ: You can create a `protected virtual` method that each class will override to do the printing, and call it from a `friend operator<<`.

FQA: Fantastic, except for the `protected` and the `friend operator<<` part. Of course you can use a plain `public virtual` method instead.

[15.12] How can I open a stream in binary mode?

FAQ: Open the stream with the `std::ios::binary` flag. That way, the stream will not translate between `'\n'` and the target platform representation of line termination, which may be different (for instance, Windows uses CRLF).

FQA: With `fopen`, pass the `"b"` character in the options string. This whole issue is pretty annoying, especially if you work with binary files on a system where `'\n'` is not actually translated, and forget to open them as binary, and everything works, and then you port the program to a system where `'\n'` actually is translated, and then you have to find all those cases and open the files as binary. However, this is not the fault of C++. It is the fault of the distributed nature of the human race, which fails to standardize the simplest things.

Many programs may screw up your binary files due to this family of issues, for instance, many FTP clients will do so unless explicitly told otherwise.

[15.13] How can I "reopen" `std::cin` and `std::cout` in binary mode?

FAQ: There's no portable way to do it.

FQA: That's probably the fault of all those different incompatible systems rather than C++ or any other programming language.

[15.14] How can I write/read objects of my class to/from a data file?

FAQ: Read the section about serialization.

FQA: The FQA doesn't have a section about a serialization. Short summary: you're in for a rude awakening. There's no standard serialization mechanism in C++. Furthermore, there's no way to define a reasonable custom one since there's no reflection (no way to figure out the structure of an arbitrary object given a pointer to it, no way to create an object of a class given its name or some other sort of ID, etc.).

However, there are many custom packages for serialization (typically thousands of source code lines, requiring you to use hairy macros/templates for each serialized class member). Or you can roll your own, or you can dump the whole snapshot of your process to a file in non-portable ways (may be the cheapest thing to do more frequently than it sounds).

[15.15] How can I send objects of my class to another computer (e.g., via a socket, TCP/IP, FTP, email, a wireless link, etc.)?

FAQ: Read the section about serialization.

FQA: Keep your expectations low.

[15.16] Why can't I open a file in a different directory such as `"..\test.dat"`?

FAQ: Because `\t` expands to the tab character. Use `\\t` to say "backslash followed by t".

FQA: You have to escape certain things somehow, so this is perfectly legitimate - unlike the fact that the C++ standard library has no way to *open a directory*, for example.

[15.17] How can I tell {if a key, which key} was pressed before the user presses the ENTER key?

FAQ: The C++ standard doesn't even assume your system has a keyboard. So there's no portable way.

FQA: Good for the C++ standard, but is there a reason not to provide a standard interface for systems which do have a keyboard?

[15.18] How can I make it so keys pressed by users are not echoed on the screen?

FAQ: You can't. See above.

FQA: Yep, some systems don't have screens - we sure shouldn't support the ones that do.

Interestingly, the C++ standard assumes that your system has persistent files, and has separate output and error streams. Even more interestingly, it assumes that you might need Unicode output and locales. You'd be surprised to find out some of the strange places where devices carrying dead code supporting these features live.

[15.19] How can I move the cursor around on the screen?

FAQ: Using whichever way that works on your system. It's not related to the C++ standard.

FQA: You should have spotted a trend by now.

[15.20] How can I clear the screen? Is there something like `clrscr()`?

FAQ: There is, but not in the C++ standard. It's system-specific.

FQA: So are windows, sockets and regular expressions. You see, the C++ standard doesn't assume you have a screen, a network controller or hardware for regular expression matching.

The lame standard library is another one of those things making using C++ a joy.

[15.21] How can I change the colors on the screen?

FAQ: Depends on your system. The C++ standard doesn't refer to this.

FQA: But there's an ANSI standard that does. There are some ugly character sequences you can send to `cout`/`stdout`; you don't need to call any special functions. Many terminals support this.

Freestore management

This page is about one of the most hard, boring and dangerous things C++ forces you to do manually - killing your objects at the right time. One dirty job, that.

[16.1] Does `delete p` delete the pointer `p`, or the pointed-to-data `*p`?

FAQ: That would be `*p`. The keyword should have been `delete_whatever_is_pointed_by`. Similarly, `free` should have been called `free_whatever_is_pointed_by`.

FQA: It really should have been "". That's right, the keyword should have been an empty string. *You* don't need it. The object should live as long as someone can use it, and when it becomes unaccessible and can no longer be used by anyone, it should die. Why is making sure that this is what actually happens *your* job?

Of course garbage collection may be time consuming in the average and/or worst case. Are you sure your implementation of `new` is better in this respect though? At least with garbage collection and managed pointers you can do heap compaction. With `new/delete` and bare pointers, pieces of memory between used blocks too small to satisfy an actual allocation request will accumulate, and you'll effectively run out of memory. This nice situation is called external memory fragmentation. Try it with your production code that's supposed to have long (not to mention "unlimited") uptime - it's fun!

And what if you make a mistake, one single mistake with all these deletions? Either you'll run out of memory, or your program will crash, or it will corrupt its own data. Which is why many experienced C++ programmers do everything to avoid explicit calls to `new`. Instead, they use RAII - have some constructor do the `new` and the destructor do the `delete`. Which eventually leads to lots of unnecessary copying - you can't point to some data inside a data structure since the data structure may be about to die, and it will kill all its data whether or not someone points to it. So you have to make a copy of that data, and keep believing that manual memory management is what makes your C++ programs so fast.

If managing the life and death of objects is such a big deal, perhaps the language isn't very object-oriented after all, is it?

[16.2] Is it safe to `delete` the same pointer twice?

FAQ: It isn't. Don't do that. Your program may crash or corrupt its own data. If it works in a test, it doesn't mean it always works. Don't do that.

FQA: When you `delete` the pointer, it still points to the same place - the only difference is that the place was marked as "free" in some system-specific way. The second call to `delete` will probably try to mark it as "free" again. Which may be problematic when it's no longer free, actually, because someone has already allocated an object there, and you've just wiped it out with your second `delete` call, so now *still other someone* can allocate an object there and overwrite the object you've destroyed so immorally.

There are other ways for this to fail - say, the code marking blocks as "free" and assuming they are taken may count on some memory right before the place pointed by the block start pointer to contain some meta-data it no longer contains, etc.

Here's what happens in managed environments. First, you don't delete anything: the environment does. Second, it never deletes anything unless nobody points to it. So you are never stuck with pointers pointing to graves of dead objects, which may already be inhibited by freshly created objects. Which is good, because whatever your project is, memory management is not one of its stated goals, is it? It's nice not to do something you don't really want to do.

[16.3] Can I `free()` pointers allocated with `new`? Can I `delete` pointers allocated with `malloc()`?

FAQ: You can't. Don't do that. Your program may crash or corrupt its own data. If it works in a test, it doesn't mean it always works. Don't do that.

FQA: The many duplicate C++ facilities, such as `new` and `malloc`, are ugly, but using mismatching functions for allocation and deallocation is not less ugly. Why do you want to do that? It's like using an opening parenthesis and a closing bracket. How can you count on something like this to work reliably?

[16.4] Why should I use `new` instead of trustworthy old `malloc()`?

FAQ: `new/delete` call the constructor/destructor; `new` is type safe, `malloc` is not; `new` can be overridden by a class.

FQA: The virtues of `new` mentioned by the FAQ are not virtues, because constructors, destructors, and operator overloading are garbage (see what happens when you have no garbage collection?), and the type safety issue is really tiny here (normally you have to cast the `void*` returned by `malloc` to the right pointer type to assign it to a typed pointer variable, which may be annoying, but far from "unsafe").

Oh, and using trustworthy old `malloc` makes it possible to use the equally trustworthy & old `realloc`. Too bad we don't have a shiny new `operator renew` or something.

Still, `new` is not bad enough to justify a deviation from the common style used throughout a language, even when the language is C++. In particular, classes with non-trivial constructors will misbehave in fatal ways if you simply `malloc` the objects. So why not use `new` throughout the code? People rarely overload `operator new`, so it probably won't get in your way too much. And if they do overload `new`, you can always ask them to stop.

[16.5] Can I use `realloc()` on pointers allocated via `new`?

FAQ: Guess what - you can't. `realloc` may end up copying memory, and C++ objects don't like to have their memory copied without getting notified. They like to have their copy constructors and `operator=` handle the copying.

By the way, why do you think `malloc/realloc/free` use the same heap as `new/delete`?

FQA: The fact that there's no `renew` or whatever you'd call it is one of the reasons to use custom allocators instead of `new`. You can then allocate uninitialized memory with `malloc` and use placement `new` to call constructors. Code doing these things is usually as ugly as sin, hard to get right and gets in the way of debuggers.

Many C++ objects will live happily ever after they're moved (`realloc`'d). A different set of objects always get broken - those *pointing to the old place*. The intersection of these sets is non-empty since objects can keep pointers into themselves. Which is the *special case* that's sort of solved by copy constructors and `operator=` (the solution is simply to have *you* implement the copying so that pointers are set up correctly).

However, the *general case* can't be solved - you can't move a C++ object and destroy the old one unless you can prove that no pointers are left to the old location. This can't be proved automatically in the general case (the halting problem and all that). Which is why you can't do automatic heap compaction, which could solve the external fragmentation problem.

If your implementation uses two heaps, one for `malloc` and one for `new`, then it stinks, because sometimes you want to somehow replace `malloc` with something else, and it's nice to be able to do it once (for `malloc`), not twice (for `malloc` and `new`), and two heaps probably mean more fragmentation, and what's the point of two heaps? I've never seen this done, but it actually is legal.

[16.6] Do I need to check for `NULL` after `p = new Fred()`?

FAQ: No, that would be bad. `new` throws an exception, so you don't have to insert tests all over the place. Unless you're using an old compiler (you can still work around it and have an exception thrown).

FQA: C++ exceptions are frequently worse than having your code simply and straight-forwardly crash (at least in the latter case, you have a good chance to find the call stack where that happened). Disabling exception support at your latest & greatest compiler is sometimes a good idea.

If you really have to write code handling out-of-memory conditions gracefully, that's not as easy a task as simply catching a C++ exception. First, you'll probably have to avoid recursion (C++ doesn't throw stack overflow exceptions, you know, so the program can't behave gracefully when you're out of *that* memory). Second, what are you going to do when you're out of memory? Take into account that you'll need *memory* to do it; you might need to reserve some in advance.

And there aren't that many choices in most cases. You can exit with an error message instead of "hard" crashing. You can show a message saying that there's not enough memory and could the user please close some programs and then you'd retry, or maybe the user wants your program to quit? All these can be done without exceptions.

Exceptions can be sort of handy when your approach is to "abort the current operation", but exceptions are not a very good way to develop robust code, and if you actually want to deal gracefully with out-of-memory situations (you probably noticed that it's rarely done), that's way out of their league. The reason is that with exceptions, there are many ways to make subtle errors in your error handling code, and when some aspect of your application is important, you're usually better off making the handling of that aspect explicit and clear.

[16.7] How can I convince my (older) compiler to automatically check `new` to see if it returns `NULL`?

FAQ: You can pass a callback function throwing an exception to `std::set_new_handler`. This will work, except for global variables calling `new` before you set your handler. There is no way to make sure your handler is set first - even if you set it in the constructor of a global variable, that variable is not necessarily the first one to be constructed.

FQA: I wonder why you want `new` to throw exceptions. This desire may be the indication of a certain kind of mindset. Hmmm, let's conduct a test: are you happy with the FAQ's solution, or are you actually bothered by the fact that *a constructor of a global variable* may run out of memory without throwing an exception?

If this option scares you, it probably means that you like to do hairy things before `main` in undefined order, including throwing and catching exceptions, which confirms my worst fears. I hope you get out of these habits by the time we happen to work on the same project.

If, on the contrary, you think that global constructors shouldn't get that complicated, your approach is apparently a little bit more practical, and now I'm really puzzled. Are you absolutely sure you want `new` to throw exceptions?

[16.8] Do I need to check for `NULL` before `delete p`?

FAQ: `delete` already does that, so no, you shouldn't! *You could get the test wrong*, and you'll spend time testing both execution paths *as required by testing methodologies*.

FQA: Apparently the FAQ is written for imbeciles that can't test for `NULL`, *and* they don't test the code after writing it to see if it worked, *but* they *do* mechanically test all branches because of *testing methodologies*. If you want to really create a vivid image of a member of the FAQ's target audience in your imagination, think about this: just *how* does the idiot cover both execution paths? The poor creature must artifically create cases where `p` is `NULL`, without having the rest of the program crash. Which can be tricky enough to be inconsistent with the rest of our data about the mental capabilities of this programmer. And now we get a clear picture: the FAQ is designed for extraterrestrial intelligence struggling with the many difficulties of C++.

Actually, there is no reason to be mean this time. Once in a lifetime those people actually made something easy. Maybe they were inspired by the example of `free`, which also accepts null pointers. Pretty inconsistent with the spirit of the language. Which is why I was very surprised when I first found this out.

[16.9] What are the two steps that happen when I say `delete p`?

FAQ: When `p` is a `T*`, the first step is `p->~T();`, and the second is `operator delete(p);`.

FQA: Ugly syntax, isn't it? Especially the fact that `a+b` is the same as `operator+(a,b)`, but `delete p` is *not at all* the same as `operator delete(p)`.

The semantics are consistent with the decoupling of *allocation* (`operator new`) and *construction* (`T()`). This decoupling breaks encapsulation without admitting it (the caller must know the size of the objects of the class at compile time, which means it must know all the `private` members) in order to increase efficiency in straight-forward implementations (which can't do just-in-time compilation with optimization and have to know the size at compile time to optimize allocation). Which is why C++ code is recompiled all the time.

[16.10] In `p = new Fred()`, does the `Fred` memory "leak" if the `Fred` constructor throws an exception?

FAQ: No, because the compiler effectively generates a `try/catch` block around the constructor call, and when exceptions are thrown, the `catch` part deallocates the memory and rethrows the exception.

FQA: If you throw exceptions in constructors, make sure they are caught when custom allocators are used (the kind that looks like this: `new (pool.alloc()) Fred()`).

If you don't throw exceptions, the implicit `try/catch` around `new` is one illustration of the fact that exceptions increase the size of your code even when you don't use them. One good thing is that most compilers have a flag disabling exceptions.

[16.11] How do I allocate / unallocate an array of things?

FAQ: You allocate it with `p = new T[N]` and deallocate with `delete[] p`. Using `delete p` is an error.

FQA: You can only do that if `T` has a default constructor (one that can work without arguments). Which is one more reason to avoid non-trivial constructors. Alternatively, if you'd rather create a problem than solve one, you can replace your evil arrays with `std::vector`.

[16.12] What if I forget the `[]` when deleting array allocated via `new T[n]`?

FAQ: You'll infect your program with an incurable decease. It will do something like corrupt its data and die.

FQA: If you want to realize the idiocy of this rule in its full glory, consider this. `new` calls `malloc` or some other allocator, and it passes it the block size. The allocated pointer is then passed to `delete`, but the size is *not*. How does `delete` know to free a block of the right size - not too little, not too much? *It has to store the size somewhere*, doesn't it? But *of course* it *can't be bothered* to figure out the number of objects pointed by the pointer (like, divide the stored size by `sizeof(*p)`, making an actual *use* of "type safety") so it can call the destructors properly.

But wait, there's more! What's the deal with *data corruption*? Shouldn't the worst possible effect be a resource leak due to the fact that some destructors are not called?

Here's the best part. `operator new[]` allocates *a little bit more memory* than it's asked for in order to *store the number of the frigging objects* in that place (it can also be done in other ways, but the effect is equivalent). `delete[]` uses it to call the destructors for all the allocated objects. Since `new` doesn't store any number and only allocates the amount of memory it was told to, it becomes clear why mismatching the `new` and `delete` operators will lead to data corruption.

The bottom line is this: C++ stores the number of objects in a block *once* when `new` is called and *twice* when `new[]` is called, *and it will rather have you introduce lethal bugs in your program than use the information to help you get it right*. Now that is what I call "hospitality".

[16.13] Can I drop the `[]` when deleting array of some built-in type (`char`, `int`, etc)?

FAQ: No you can't. You may think that you can, because `int` has a trivial destructor. But it's not just about destructors. For example, what if someone replaces `operator new[]` and `operator delete[]` in a way incompatible with mismatching `new/delete[]` calls?

FQA: Yeah, did you think about that? And what if you use an array of `Int`, which is a `typedef`, and someone changes the `typedef` to `SmartIntClass` instead of plain old `int`? Think about it. That's what your brain is for: thinking about exciting, useful things like this.

The serious answer is that if a language has two sets of similar matching operators, than a programmer is better off avoiding mismatches between those operators. The fact that there really should have been one `delete` operator, no, wait, make it zero, may be a reason to switch to a different language, but not to violate the rules of a language.

[16.14] After `p = new Fred[n]`, how does the compiler know there are `n` objects to be destructed during `delete[] p`?

FAQ: Sing along: "It's a kind of magic..."

There are two common ways used by our friends the compiler-implementing magicians. One is to allocate extra memory and store the number of objects there. The other one is to use an associative array mapping from pointers to sizes.

FQA: You may wonder why these clever magicians don't rely on `malloc` to do the magic, and have you end up with the block size stored *twice*: you ask `new` for 8 bytes, then `new` asks malloc for 12 bytes, then `malloc` asks `sbrk` or whatever's down its guts for 16 bytes.

The answer is of course *modularity*. Translation: why should the implementor of the C++ language bother to figure out the details of `malloc` when there's the easier option - leave it to the implementor of the C language and simply and portably implement `new[]` on top of the C runtime? 4 bytes of your memory are surely not a good enough reason.

"Magic". The next thing you know, they'll call exploiting security holes in your OS to utilize some of those unused processor & memory resources "magic".

[16.15] Is it legal (and moral) for a member function to say `delete this`?

FAQ: You can do this, as long as you are sure the object is allocated with `new`, and the pointer is not used for anything at all after `delete this`.

FQA: This is a bit weird, and it probably causes many people to get alarmed and start anxiously looking for places that *might* touch the deleted object (as if it were more likely than a less exotic access to a dangling reference). And it forces the user of a class to allocate the objects with `new`, which is against the (misguided) spirit of C++ and therefore another source of confusion. At least provide a `static` method that allocates objects with `new` and declare the destructors `private` to document your intent to control the allocation.

Why do you want to do it anyway? If you want to impress people, why not do some real magic - actual functionality, not just yet another syntactic exercise?

[16.16] How do I allocate multidimensional arrays using `new`?

FAQ: The answer is very long with 3 large code listings.

FQA: There is no `operator new[][]` in C++ - it decided that `new` and `new[]` are sufficient; it had many stupid reasons to decide that way. Anyway, you have two options: allocate a flat array and manage the N-dimensional indexing manually (generating one-dimensional indexes using expressions like `x+y*width`), or you can allocate an array of arrays *(correction)*. For 2 dimensions, you need to allocate a `new T*[M]` and then in a loop with `M` iterations allocate a `new T[N]`.

With the second way you get more natural-looking indexing, but worse performance (less speed, more space). It may be convenient if you need each one-dimensional array in the two-or-more-dimensional array to have a different length, but even then it's probably better to fake it with a flatter kind of array in ugly ways if you care about performance (by the way, if you don't, make the best of it - try a safe language).

[16.17] But the previous FAQ's code is SOOOO tricky and error prone! Isn't there a simpler way?

FAQ: Sure it is! You can define a `Matrix` class and have it do all the dirty work. Which is good, as should be obvious to anyone familiar with *Star Trek 2* (a reference follows).

FQA: I didn't see *Star Trek*, but moving the dirty work to a single place doesn't make that work "simple". Of course it's better than doing that work over and over again, but how many multi-dimensional array classes have you seen, and what happens when someone tries to convert between them? I mean it can't be a "single place" unless that place is the compiler.

By the way, if you need performance, you can't really encapsulate the layout of the multi-dimensional array, because you'll need to do things like get one column to the right (with `flat_index+1`) or one row up (with `flat_index+row_width`), instead of computing things like `x+(y+1)*row_width` every time through the innermost loop.

You can try to define all kinds of functions like `increment_by_row`. If you really believe that it will make it possible to "transparently" change the representation later (thus turning all the optimizations carefully developed for this representation into pessimizations), go ahead and define and optimize hordes of such tiny functions and keep lying to yourself about how this makes your code more readable.

Data representation is very important. Too bad so many people believe that the only important thing about *data* is to hide it behind *code* (interfaces) so that you can "always change the data representation later". While this may be right in the majority of cases, it is likely to be wrong in *the most important* cases.

[16.18] But the above `Matrix` class is specific to `Fred`! Isn't there a way to make it generic?

FAQ: Of course! You can use templates!

FQA: This way, not only will your code run slowly - it will also compile slowly.

[16.19] What's another way to build a `Matrix` template?

FAQ: You can use `std::vector` instead of bare pointers!

FQA: You can also use any of the numerous array classes implemented before `std::vector` was introduced into the C++ standard library (which was long after C++ became widely used).

Alternatively, you can also use any of the numerous *matrix* classes implemented before `std::matrix` was introduced into the C++ standard library... Actually, make that "before `std::matrix` *will be* introduced into the C++ standard library". Can you explain why the C++ standard includes `std::vector` - a class quite similar to built-in arrays, but different - but does not include `std::matrix`, a class that would be quite similar to built-in 2D arrays, but different? Here's a more difficult question: is it good or bad that we don't have `std::matrix`, provided that we *do have* less than optimal built-in 2D arrays? Before you answer, consider type conversions, dynamic resizing, literal values, telling the size of an array at run time...

That's what C++ is all about: the development of your ethical judgment. Everybody can tell good from bad; choosing between the ugly and the disgusting takes real wisdom.

[16.20] Does C++ have arrays whose length can be specified at run-time?

FAQ: That would be yes - there's `std::vector`.

Actually, it's a no - you can't do it with built-in arrays.

Wait a minute - you *can* do it with *the first index* of a built-in array, for example: `new T[run_time_val][compile_time_val]`.

You know what? Arrays are evil.

FQA: `std::vector` is not an array, it's a class that looks like an array until you try to initialize it with `{1,2,3}` or pass an array to a function expecting a `vector` or get a huge compiler error message. Built-in arrays don't behave like this.

C++ inherits the following rule about types from C: the size of a type is always known at compile time, and you can get it with `sizeof`. This rule is useful in C because, *together with other rules* which are violated in C++, it makes it quite easy to mentally map source code to assembly code (sort of) and estimate run time performance. In C++ it's no longer useful, because telling the *meaning* of C++ code is nearly impossible, not to mention reasoning about its performance.

Anyway, the length of a built-in array is part of its type, so if it could be dynamic, it would violate the ex-useful rule. So it can't. However, built-in *pointers* can point to an array of length defined at run time. So you can have a `T* p` pointing to `run_time_val` objects, but you can't have an array of type `T[run_time_val]`. Which means that you can't allocate such arrays on the stack (unless the compiler decides to *almost* violate the ex-useful rule and support it, the way GNU C/C++ does, for example).

The built-in C++ arrays are inherited from C without changes (well, except for constructor/destructor calls and `new` in addition to `malloc` and other stuff of this kind), so if you care about details, the best place to look is material about C. This is one of the many examples illustrating that you have to know C in order to really understand C++.

[16.21] How can I force objects of my class to always be created via `new` rather than as locals or global/`static` objects?

FAQ: You can define the constructors `private` and provide `public` functions which return pointers to objects allocated with `new`. Which goes by the fancy name of "The Named Constructor Idiom".

FQA: Why do you want to force this? Allocating locals is more efficient than `new`, and you wouldn't have to worry about `delete`, either. The cost is that when you change the `private` members of the class, all code using it gets recompiled.

Wait, is that what you want - to avoid recompilation? In that case, the "Named Constructor Idiom" defeats the purpose - change a `private` member and you still trigger a rebuild. You can either drop C++ classes and store the state in a C `struct` (place a forward declaration in the header file and the full definition in the implementation file), or you can wrap this exact solution with the code of an extra C++ class just because C is evil. The latter way is known as "The Pimpl Idiom" ("pimpl" apparently stands for "pointer to implementation").

Idioms make the world move. Especially the Design Pattern Idiom. Or the Idiom Design Pattern. Or something.

[16.22] How do I do simple reference counting?

FAQ: Two large code listings are given (a quote from these listings: `// DO NOT CHANGE THE ORDER OF THESE STATEMENTS!`), followed by a bold claim that now you have a way to do simple reference counting.

FQA: You do **simple** reference counting by using a language which does reference counting (or garbage collection - you probably don't care if you want it "simple").

You do **complicated** and **broken** reference counting by using C++, creating smart pointer classes (and smart array classes, and smart multi-dimensional array classes), wrapping your private constructors with public functions returning smart pointers to objects (or smart arrays of objects) to make sure all objects are actually pointed by smart pointers so the reference counting is not entirely worthless.

Congratulations! You've just wasted lots and lots of time to emulate garbage collection on top of C++. Well, the code doesn't compile as fast as it would with built-in garbage collection (all those smart pointer templates), and the error messages are a bit cryptic (all those smart pointer templates), and there are those tricky cases like cyclic references you still fail to deal with (despite all those smart pointer templates), and for every class you write extra code wrapping constructors, and the whole thing doesn't play well with existing libraries (which use *different* smart pointer templates), and...

Why do you want to do reference counting in C++?

[16.23] How do I provide reference counting with copy-on-write semantics?

FAQ: You can have your class keep all the data in a structure pointed by a member pointer `_data`, and then each method that wants to modify the object has to check the reference count, and if the data is shared by several objects, the data must be cloned before the modification.

FQA: Ultimately, what are you trying to achieve? If you care a lot about efficiency *and* you care a lot about high level of abstraction *in the same piece of code*, there's probably no easy solution for you. Are you sure it's *the same* code? Or maybe you want efficiency in some places, and high level of abstraction in other places? Then you can use two different languages.

If it's really the same code, the best solution is probably to write your own little language. It's not as hard as it sounds, and not so strange - think about it: apparently you must write a lot of code (if there wasn't a lot of it, it wouldn't be a problem) in some pretty specific domain, because there's no high level language which supports the right built-in facilities. Or is there? Are you looking for copy-on-write strings or something? *String processing in C++?* Do yourself a favor and stop right there. Use a language with decent built-in strings.

If there really is no good existing high-level language for your job, basically what you have to do is meta-programming: you want something which is not directly related to the specific meaning of your program - you want objects of pretty much arbitrary classes to behave in a certain way. Which means you want a new language. You can do it with your own compiler or you can do it in a system which supports meta-programming well (the two approaches are really pretty close). C++ meta-programming is a nightmare - the facilities for it are very poor, and there's lots of strange features already in the language that you must interoperate with. You have been warned: this path leads to the gates of madness.

Clarification: I'm not saying that meta-programming is likely to be the solution when you want copy-on-write. On average, that would probably be over-engineering. However, in the cases where it indeed is over-engineering, so is the FAQ's solution; the way to go is probably to have the user code explicitly copy objects upon modifications. And when there's enough code involved to make that tedious, then the C++ solution also gets tedious, and that's when meta-programming could be appropriate.

[16.24] How do I provide reference counting with copy-on-write semantics for a hierarchy of classes?

FAQ: You do it by writing a lot of code! Like this:

```
code;
code;
code;
lots of code;
more code;
two screens of code;
//boy is this FUN!
code;
```

FQA: Here's a solution for another hierarchy of classes (you might have more than one in your project - it happens, and the FAQs "solution" is done on a per-method basis, and *there isn't even a way to iterate over the methods of a class* in C++, not to mention "transparently instrument the methods of a class with custom logic like copy-on-write"):

```
code;
```

```
  code;
  //why am I doing this?
  code;
  code;
  //help
  code;
  //I think I'd rather become a farmer
```

Seriously, I'm not going to discuss the nightmare that is the FAQ's proposed solution. Follow the link to the FAQ's answer if you want to check it out. What I would like to point out is that it's possible to do all this transparently if your environment has good support for meta-programming. For example, some languages let you write your own code to implement object attribute modification (the simpler option), and/or automatically generate wrapper classes to intercept methods changing objects (the more complicated option). Either way, you end up with O(1) code to solve the problem instead of O(N), where N is the number of classes involved.

[16.25] Can you absolutely prevent people from subverting the reference counting mechanism, and if so, *should* you?

FAQ: You can't, and you *usually* shouldn't. (Yes, that's pretty much what the FAQ says: sometimes you *should* do what you *can't*. Just what does "should" mean in its warped universe?)

There are two holes in the armor. First, your `SmartPtr` probably has an `operator*` that returns a bare reference to an object. So you can end up with `Dumb* p = &*smart_p;`. This can be sort of closed using several approaches, each worse than the others in its own unique way. And then there are ways to get the bare pointer with syntax like `smart_p.operator->()`, returning a `Dumb*`.

The second hole is that someone can have dangling references to `SmartPtr`. This can't be prevented even by returning a `SmartPtrPtr` in an overloaded `operator&`, because C++ has *references* (remember - the new feature you should use instead of the evil pointers), and there's nothing you can overload to prevent someone from taking a reference to your `SmartPtr`.

Use code reviews to figure out what's actually going on with all those smart pointers.

FQA: You can't fake what you don't have (I think the quote is attributed to S. Cray). C++ doesn't have safe memory management. Faking safe memory management leads to still unsafe memory management, with the important bonus of obfuscation.

You can absolutely prevent people from subverting the reference counting mechanism by using a programming language that absolutely prevents people from subverting its built-in memory management mechanisms.

[16.26] Can I use a garbage collector in C++?

FAQ: Sure you can. Let's compare it to smart pointer techniques. Garbage collection is not as portable, but typically more efficient, and it can handle cyclic references, and it works better with others' libraries because you don't need to explicitly change the pointers from dumb to smart or anything like it. However, sometimes objects can leak - a C++ garbage collector can't tell a pointer from a random bit sequence that just happens to look like a pointer.

FQA: Assuming you live in a free country, what's there to stop you from using a garbage collector in C++? Sure, there are minor obstacles, for example, it doesn't really work, but it's still a free country, isn't it?

Suppose you can use garbage collection, which means that you are not worried about interference with real time requirements. *Why are you using C++?* Why not use a safe language instead? Look - the C++ FAQ admits that garbage collection is *more efficient* than C++ "smart pointers" all over the place, and more correct (cyclic references), and the single correctness problem it mentions (random bit patterns) vanishes in a managed environment. So what's the problem?

If you think you need to use C++, deal with the fact that it doesn't have garbage collection, and you must manage memory manually. Or you can deal with the other facts. For example, what happens if two libraries relying on *different* garbage collectors are supposed to work *in the same program*? What happens if code incompatible with garbage collection (because it does "clever" things with pointers, like `one_based_array = (int*)malloc(arr_size)-1`)? What about destructors? C++ garbage collectors don't know what they're freeing, so you can't have finalization functions. What about all those happy custom allocators C++ programmers adore so much? What if a `malloc`ed object points to an object allocated from such a custom pool? The destructor of the outer object with the pointer is not going to be called - so who is going to deallocate the pointed object from the pool?

C++ does not feel pain. It can't be reasoned with. Starting a fight is a big mistake. If you want to use C++, you must learn to love it the way it is, in particular, manage your memory manually.

[16.27] What are the two kinds of garbage collectors for C++?

FAQ: There are *conservative* and *hybrid* garbage collectors. The conservative ones just look for bit sequences looking like pointers. The hybrid ones require the programmer to specify some layout information explicitly in the code (but they still traverse the call stack conservatively when they look for pointers).

Garbage collectors may cause memory leaks when a bit pattern looking like a pointer is misinterpreted as a proof that the block pointed by this "pointer" is still in use. And some illegal programs may "confuse" garbage collectors (that's the word used by the FAQ) by keeping pointers outside of allocated blocks. Why can't these programmers behave reasonably?

FQA: Let's start with a different question: what's the problem with garbage collection in C++? Answer: there is no way to inspect the state of a C++ program and tell which blocks are in use. In environments designed to support garbage collection, you can do that by checking if the program can reach the block using one of the pointers it currently keeps in its memory. But in C++, you can't. First, you don't know where the program keeps pointers (no real run time type information), and second, pointers that don't point into a block can be used to compute pointers that do point into that block (unsafe pointer arithmetics).

So along come the memory leaks, and the "confusion". And what *happens* when a garbage collection is "confused"? I'll tell you what happens - it frees an object which is still in use. The result is a crash or a data corruption. Do you think "confused" is a legitimate euphemism in this context? "Oh, I'm so confused! I think I'll kill your program now."

As to the "unreasonable" programmers - well, adjusting a pointer to point before an allocated block is one efficient way to implement one-based (instead of zero-based) arrays, among other things. Of course you can allocate an extra element and avoid violating the rules. But most people don't know these rules, because although they are a part of the language definition, they are not enforced in the widespread implementations, and don't become a part of the mental model developed by programmers as they gain familiarity with the language.

That is, not only does it *work* when one implements one-based arrays this way - one *doesn't see a reason for it not to work*: experience consistently tells people that C++ pointers are just a kind of integer. Formally, there are reasons for this to be illegal (like, duh, what happens if you want garbage collection?) - but most people are not language lawyers. In practice, what matters is the de facto standard (that's why it's called "de facto").

[16.28] Where can I get more info on garbage collectors for C++?

FAQ: Here.

FQA: Don't bother. If garbage collection worked in C++, you would hear about it. C++ has been around for decades, there are hundreds of thousands of C++ programmers, and megatons of C++ code with zillions of memory management bugs. If someone implemented a working solution for this problem, they'd become rich, famous, or both, and the garbage collector(s) would be everywhere.

Why do you think you don't see too many around?

Exceptions and error handling

This page is about C++ exceptions - an error handling facility which may be worse than dereferencing a null pointer upon error.

[17.1] What are some ways `try / catch / throw` can improve software

quality?

FAQ: You'll have less `if` statements in your code: you won't have to check for errors each time you call a function. Conditional statements are known to contain more bugs than other statements. With less `if` tests, you'll ship a better product, faster.

FQA: This is cargo cult programming. Conditional statements are error-prone because they are used to handle complicated scenarios, where an action can result in many different outcomes, which affect the next actions. In order to make errors less probable, one has to simplify the model of the desired behavior of the software. The problem is the complexity that leads to `if` statements, not the `if` keyword, and using different keywords is not going to solve the problem by itself.

Exceptions are supposed to simplify the error handling model based on the assumption that in most cases, a function that detected an error can't handle it, and has to propagate it to the caller. Finally, a "high-level" enough caller is reached and actually makes a decision (pops up an error message, tries a different action, etc.).

Despite its promises, this approach has inherent problems. There's a "social" problem - with exceptions, people are not aware of the different errors that may happen in the code because most of the code doesn't deal with errors. And when people rarely think about a particular aspect of an application, ultimately this aspect is unlikely to be handled well. There's a more "technical" problem - functions essentially doing nothing upon error except for propagating errors to the caller still can't be completely unaware of errors. That's because they may need to release the resources they acquired before returning to the caller, which may lead to *more* errors, which must also be handled. Finally, in practice exception support has run-time overhead, and very significant code size overhead, even if exceptions are never raised at run time, and even if they are *not mentioned* in your code. C++ devotees may claim otherwise; you can check by compiling your code with and without exception support (if your compiler doesn't have such a flag, compile code as C and as C++ instead). This is unacceptable in resource-constrained systems.

Still, in many cases, the benefits of exceptions are more important than their problems. For example, if your language manages memory automatically, the problem of releasing acquired resources becomes a small one (you only have to care about files, etc., which are a tiny part of the "resources" used by a program - most of the "resources" are memory). If your language throws exceptions when you violate its rules (for example, upon out-of-bounds array access), these exceptions will help you find lots of bugs, especially if you can get the call stack from an exception. If the purpose of an application is automated testing, and/or it's used as a quick-and-dirty internal tool as opposed to a product for an end user, this kind of exceptions is all you need to handle errors of almost all kinds. In some languages, you can even resume the execution from the point where the exception was raised after fixing the problem at the point where it was caught.

C++ exceptions offer none of these features. "Exception-safe" C++ code can't handle errors which happen when it tries to release resources; "exception-unsafe" C++ code will leak resources, most frequently memory; and once you throw an exception, the call stack is lost. This means that even separating your code to several processes and executing code like `*(int*)0 = 0;` upon error is a better way to handle errors than C++ exceptions: at least the memory is going to be reclaimed by the operating system, and you can typically have it save a snapshot of the process, so that you can open it in a debugger and see where the error happened. A recommendation to "ban" exceptions is probably over the edge, but think *a lot* before using C++ exceptions, *or* a feature that implicitly depends on them, such as constructors and overloaded operators, which have no other way to report an error. What C++ calls "exceptions" is as unlikely to give you the benefits people get from exceptions in other languages as what C++ calls "classes" is unlikely to give you the benefits of OO.

[17.2] How can I handle a constructor that fails?

FAQ: As you'd guess from the location of this question in the FAQ, the answer is "by throwing an exception". Alternatively, you can mark the object as a "zombie" by using some kind of validity flag. You can then check that flag in the calling code and maybe in the member functions of the object. The latter solution tends to "get ugly".

FQA: The inability to gracefully handle errors in C++ constructors is one good reason to avoid constructors that do more than nothing, and use initialization functions instead. And C++ exceptions are not a graceful way to handle errors, *especially* in constructors. If your member object constructor throws an exception, and you want to catch it in your constructor, the normally ugly colon syntax gets much uglier.

By the way, the C++ standard library doesn't throw exceptions in constructors (except for the ones thrown by `operator new`). For example, you are supposed to test whether `ofstream` objects are zombies when you pass them a filename in the constructor.

[17.3] How can I handle a destructor that fails?

FAQ: Actually you can't - not beyond logging the problem to a file or the like. In particular, *do not* throw an exception. The problem is that destructors are called when exceptions are thrown so that functions propagating errors to their callers can clean up resources. Your destructor can also be called in such a situation. And when an exception is already thrown, throwing another one will result in a call to `terminate()`, killing your process. Because you see, what else could C++ do? There's an ambiguity: which exception out of the two do you want caught now?

Strictly speaking, you can make that "do not throw exceptions in a destructor unless you are sure that it won't be called as a result of an exception already thrown", but you can rarely be sure of that.

FQA: That's right, `terminate()`. Solomon-style conflict resolution carried to the end. See? Exceptions are not a graceful way to handle errors.

And "don't throw exceptions in destructors" actually means "don't call functions in destructors unless you are sure they don't throw an exception". The C++ compiler won't check for you, because it can't: the language doesn't force a function to declare whether it throws exceptions.

This is one good reason to avoid destructors doing more than nothing: like constructors and operators, they can't handle errors.

[17.4] How should I handle resources if my constructors may throw exceptions?

FAQ: Well, the destructor of your class will not get called, but the destructors of the successfully constructed sub-objects will get called. Conclusion: you should have all the resources allocated by your constructors assigned to sub-objects. For example, if you call `new` in a constructor, don't use a bare member pointer to hold the result - use a smart pointer, like `std::auto_ptr`. You can also define your own smart pointer classes to point to things like disk records! Groovy!

And you can use `typedef` to make the syntax of using smart pointers easier.

FQA: **WARNING** - cyclic dependency between C++ features detected! You see, exceptions are a must in this language so that you can handle errors in all the functions which fail to look like functions, such as constructors & operators. Then it turns out that you need *constructors* to work with *exceptions* - unless each and every piece of memory you acquire is not immediately assigned to some smart pointer, your code is not exception safe. This is known as "Resource Allocation Is Initialization" (RAII) in the C++ community; it's supposed to be a *good* thing.

And smart pointers are no picnic, as are virtually all automatic devices with something like "smart", "simple" or "fast" in their name. Sure, you can use `typedef` to simplify the syntax. So can someone else; you'll end up with many different type names for the same thing. This may annoy people, but it's perfectly OK with the compiler - when it spits an error message, it simply substitutes the full type names for all `typedef` names. But you can write a program to filter the error messages...

Seriously, the syntax of smart pointers is the small problem. The big problem is their semantics. When you see a bare pointer, you know how it works. But a smart pointer can work in *a lot* of ways. The boost libraries allow you to instantiate hundreds of different smart pointer classes from a single template (which made it to TR1, so we're going to see it in the next version of the C++ standard). How are you going to figure out whether your program manages resources correctly or not when it's littered with smart pointers of different kinds, especially in case there's any non-trivial scenario there, like the cases when "ownership" (the right & duty to dispose a resource) is passed from object to object, or there are cyclic references in your code, or whatever?

When every single piece of software is "smart", and you can't trust things like `*p` and `p->x`, the software becomes unmanageable.

[17.5] How do I change the string-length of an array of `char` to prevent memory leaks even if/when someone throws an exception?

FAQ: If you want to work with strings, use something like `std::string` instead of `char*`. Otherwise, there's lots of exceptions to catch, and lots of code to manage memory.

FQA: The FAQ is right about one thing - `char*` is a nasty kind of string, and using it for text processing is very tedious. If you're doing anything not entirely trivial with strings, `std::string` is better than `char*`; using a different language than C++ for text processing, one with a good built-in string type, is still better.

However, the part with exceptions really comes from `operator new`, not from `char*`. You can use `malloc` instead, or configure your compiler to disable exceptions.

[17.6] What should I throw?

FAQ: C++ allows you to throw objects of arbitrary types; however, you probably shouldn't throw objects of built-in types. For example, you can derive all your exception classes from `std::exception`, and throw temporary objects of your classes.

FQA: Yep, C++ allows to throw anything. Too bad you can't really *catch* it later. The only way to catch an arbitrary exception is to use `catch(...)`, which doesn't let you find out what was thrown from where, and will even catch *illegal memory access* on some systems. This makes finding code like `throw "C++ is so grand - you can throw anything!!";` a lot of fun (you have to find it on occasions when the uncaught exception crashes your program).

The FAQ's advice is thus a good one, as opposed to the language decision to allow to throw anything - a typical example of the twisted notion of "generality" used throughout the language design. This decision is completely incomprehensible unless you realize that there's a basic axiom in C++: the language must not force the compiler writer to treat any class specially. For example, having a common base class for all user-defined classes which have at least one `virtual` function could be quite handy, but it's incompatible with this implicit axiom. What did the C++ designers gain from following this bizarre rule? Apparently nothing, except for an illusion of "generality", whatever that means.

[17.7] What does `throw;` (without an exception object after the `throw` keyword) mean? Where would I use it?

FAQ: It means "throw the last caught exception". It may be handy to catch an exception object, add some context information and rethrow it; this way you get something like a stack trace. This feature also allows you to factor out several exception handlers into a function called from a `catch(...)` block. Inside the function, you list the handlers for various special cases and prefix them with `try { throw; }`.

FQA: Rethrowing the last exception is a useful feature, and many languages have it. It would be equally useful in C++ if C++ exceptions were any good. In particular, having to use this kind of feature throughout the code to get a *call stack* is an insult to the language user. Unless it's some kind of "logical" call stack (context information not equivalent to the list of C++ functions you'd see in a debugger at the point where the exception was thrown), call stacks should be provided by the language.

If you are using C++ and want to figure out the current call stack, it may be better to rely on platform-specific tricks (reading the frame pointer using inline assembly and traversing the linked list pointed by it, then translating the instruction pointers using a symbol table) than to litter your code with statements duplicating the information that's already there.

[17.8] How do I throw polymorphically?

FAQ: Suppose you have a `BaseEx` exception class and a `DerivedEx` exception class, which is inherited from `BaseEx`. Than the following code might not work as you expect:

```
void f(BaseEx& e)
{
  throw e;
}
void g()
{
  DerivedEx e;
  try {
    f(e);
  }
  catch(DerivedEx&) {
    std::cout << "derived exception caught" << std::endl;
  }
}
```

The program will *not* enter the `catch` block because you didn't throw polymorphically. That is, the statement `throw e;` throws the object `e` as a `BaseEx`, because that's the type of `e` in that context; once an object is thrown as a `BaseEx`, it will not get caught as a `DerivedEx`. If you prefer the other behavior, you can "easily get it" by having a `virtual void raise() { throw *this; }` in your base class *and* your derived class, and calling `e.raise();` instead of `throw e;`. This way `DerivedEx::raise()` is called, and in the context of that function `e` is of type `DerivedEx`.

FQA: Let's see. You use C++ exceptions. Moreover, you have a hierarchy of exception classes. Moreover, you *pass exception objects to functions*, in a way relying on an *implicit upcast*. Looks like you have lots of confidence in your knowledge of C++ features. But along comes C++ and beats your common sense once again. The startling inconsistency of the language is almost a virtue: maybe this time you will learn the value of simplicity and write something readable.

The behavior of `throw`, which looks at the static type of its argument expression, is somewhat surprising considering the behavior of `catch`, which does "respect" inheritance (to the extent made possible by `throw`). In practice, it is probably better to remove some of the complexity in the example rather than add more complexity by mixing the dynamic binding of `virtual` with the static binding of `throw`. A human might need to understand the code, you know.

If you do want to memorize the quirks of C++, try to warp your mind to think in terms used by the compiler construction peanut gallery. From this perspective, the behavior of `throw` and `catch` *is* consistent: both only look at things known at compile time (the relationships between classes), and ignore things only known at run time (the actual type of an object). Basically all of C++ behaves this way except for `virtual`, `dynamic_cast` and `typeid`. I think.

[17.9] When I throw this object, how many times will it be copied?

FAQ: Zero or more. There's no universal answer. The compiler has to make sure a thrown object provides a copy constructor, even if it doesn't actually copy anything.

FQA: If you care about performance, C++ exceptions are probably no good for you. Exception support translates to a huge mountain of code in your executable, and slows down function calls throughout your program. If you didn't care about performance, you wouldn't ask this question. If you *think* that you care about performance, but never actually measure it or look at the performance implications of the techniques you use in your code, feel free to entertain yourself with any fake answer that suits your emotional needs.

[17.10] Exception handling seems to make my life more difficult; clearly *I'm* not the problem, am I??

FAQ: *Of course* you can be the problem!

Here are some habits that may prevent you from utilizing the power of C++ exception handling:

- *Return codes style*: people may put try blocks around every function call as if exceptions were error codes.
- *The Java style*: using try/finally instead of RAII, cluttering the code with clean-up statements.
- *Organizing exceptions around the location of the error rather than its reason*: this way you need to handle the same error many times and convert between types of exceptions.
- *Categorizing errors using data members of exception classes rather than types*: this way you catch more errors than you can handle in the given `catch` blocks, and rethrow exceptions after a test.
- *Using different exception classes in different subsystems*: the FAQ decides to repeat the point before previous, probably to create an impression that there are so many wrong mindsets out there.
- *Using bare pointers instead of smart pointer classes*: a special case of avoiding RAII, admits the FAQ, but it can't fight the temptation to list yet another "wrong mindset".
- *Confusing bugs with run time errors*: if someone passes a null pointer to a function when that's illegal, the code must be fixed. Throwing an exception without fixing the calling code doesn't solve the problem.

There are other wrong mindsets as well.

FQA: Yeah, you know how it is with those humans. They always fail to realize *they* are the problem, and keep asking the wrong questions. You give them a helpful and powerful language, and all they do is shooting themselves in the feet. Clearly it's their flawed minds that must be fixed.

Let's look a little closer at the impressive list of "wrong mindsets" compiled by the FAQ:

- *Return codes style* is surely a wrong way to use exceptions, but it's one of the best ways to use *C++ exceptions*, because otherwise the chance to manage resources correctly is not very high. Of course using real return codes is somewhat cleaner, but what if exceptions are thrown by third-party libraries?
- *The Java style* is suited quite well to a managed environment like Java, because most of the so-called resources are in fact *memory*, and you only need try/finally blocks for stuff like files, of which there are few. This style is problematic in C++ which lacks garbage collection, but so is RAII which forces you to wrap everything with "smart pointers" in a language which only has built-in dumb pointers. The truth is that with C++ exceptions, you can't win.
- *Organizing exceptions around the location of the error rather than its reason*: suppose you have 2 parsers in your system. Would you prefer a single `ParseError` exception, or 2 separate classes, `HtmlParseError` and `ConfigFileParseError`? Two different modules may raise exceptions for *similar* reasons, but rarely *the same* reason - or else why are they different modules? An obvious exception to this rule are errors detected at the language level, things like `ArrayIndexOutOfBoundsException`. Luckily, the C++ language run time environment does not detect errors, so we don't need to discuss this kind of thing.
- *Categorizing errors using data members of exception classes rather than types* is not necessarily bad. C++ is centered around the assumption that you can represent almost anything using types (class hierarchies or template "pattern matching") more than any other popular statically typed language. However, this assumption is frequently refuted by reality, and sticking to it is counter-productive in many cases. For instance, if a subsystem throws exceptions of a single class, and you figure out the kind of error from looking at the object members or even inspecting error strings, you may end up with less problems compared to the case where you have several dozens of classes in your exception inheritance tree. This is a special case where modeling an aspect of your program using the language static type

system may or may not be optimal.
- *Using different exception classes in different subsystems*: WOW, this is so *nasty*! Of course all subsystems should share exception classes and other important common types. All subsystems of all systems in the world should be written in close coordination between the implementers. This way, we'll finish the Tower of Babel in no time! That's why a language should have no good way to pass an object of arbitrary type through a "subsystem". Trust us: we are the people advocating a language without a *common string type*. We sure know a thing or two about the issue.
- *Using bare pointers instead of smart pointer classes*: how about staring with the language and the standard library? When `operator new` returns a smart pointer, and `"abc"` has the type `std::string`, we'll have something to discuss. Until then, why should anyone manually emulate a high-level language on top of a low-level one throughout one's code?
- *Confusing bugs with run time errors*: this has absolutely nothing to do with exceptions. It's equally applicable to any run time error handling strategy. Unless, of course, your environment throws exceptions upon events like null pointer dereferencing, which C++ doesn't.

[17.11] I have too many try blocks; what can I do about it?

FAQ: Maybe you have a "return codes mindset" even though syntactically you use exceptions. There are many special cases of this problem in which you can organize the code differently to reduce the amount of try blocks (the FAQ lists several cases). If you can't solve the problem yourself, get a mentor.

FQA: Alternatively, you can stop throwing C++ exceptions so you won't have to catch them.

Const correctness

This is about the `const` keyword, which makes you write your program twice (or more, depending on your luck).

[18.1] What is "`const` correctness"?

FAQ: Oh, that's a great thing. You declare an object as `const` and prevent it from being modified.

For example, if you pass a `std::string` object to a function by `const` pointer or reference, the function won't modify it and it won't be copied the way it happens when the object is passed by value.

FQA: Interesting. What about vector of pointers to objects? Let's see. If the vector itself is declared `const` (as in `const std::vector<T*>`), then you can't modify the vector, but you can modify the objects. If the pointers are declared `const` (as in `std::vector<const T*>`), then you can modify the vector, but not the objects. Now suppose you have a vector of non-`const` objects, and you want to pass them to a function that accepts a `const` vector of `const` objects. Oops, can't do that - the vectors are two different unrelated types as far as the compiler is concerned (no, C++ weenies, it actually doesn't make sense, think again). You'll have to create a temporary vector of the right type and copy the pointers (which will compile just fine since `T*` is silently convertible to `const T*`).

That's what `const` correctness is all about: increasing the readability of your program and protecting you from errors without any performance penalties.

By the way, not using `const` in C++ is quite likely *not* a very good idea. Of all questionable C++ features, `const` probably does the most visible damage when avoided. Any piece of code using `const` forces every other piece touching it to use `const`, or else you won't be able to work with the objects it gives you. And here's the funny part: every piece of code *not* using `const` forces every other piece touching it *to not use it, either* - or else you won't be able to pass objects to it. So if you have one `const` particle and one anti-`const` particle, there's a big shiny explosion of `const_cast` in your code. Since `const` is hard-wired into the language (no way to pass a temporary to a function that gets a non-const reference, for example) and the standard library (`iterator` and `const_iterator`), using `const` is usually a safer bet than avoiding it.

[18.2] How is "`const` correctness" related to ordinary type safety?

FAQ: It's one form of type safety. You can think about `const std::string` as an *almost* separate type that lacks certain operations of a string, like assignment. If you find type safety useful, you'll also like `const` correctness.

FQA: It's related to ordinary type safety in a pretty complicated way. `const` and `volatile` are special cases in the type system - they are "type qualifiers". The relation between a qualified type and a non-qualified type is different from any other relation between types in the language. This is one of the very many complications in the implicit conversion and overload resolution rules.

It works smoothly for the simple cases, especially if there are no other complications. It breaks pretty hard whenever you have a pointer-like object. Pointers can get *two* const qualifiers, one for the pointer and one for the pointed values. This is awkward and unreadable and when you have pointers to pointers and three const qualifiers, you may need cryptic explicit casts, but at least there's syntax for all the levels of constness. With "smart pointers" (the things you should stuff into every possible hole because pointers are evil), there's no such syntax. That's why we have `iterator` and `const_iterator` - saying `const iterator` says that the iterator is immutable, but not what it points to. Exercise: implement a vector-like class that can get the storage pointer from the user, in a const-correct way that supports attaching both to constant and mutable storage areas.

And of course a vector of `const` pointers is compiled to a different bulk of (identical) assembly code than a vector of mutable pointers. At least here the compiler is writing the same program twice, which is better than having to do this yourself. Which also happens - `const_iterator` is one family of examples.

[18.3] Should I try to get things `const` correct "sooner" or "later"?

FAQ: As soon as possible, because when you add `const` to a piece of code, it triggers changes in every place related to it.

FQA: That's right. Since you can't get out of the tar pit, the best strategy is to climb right into the middle and make yourself comfortable from the beginning. No kidding, I actually agree with the FAQ. See also the advice above about *not* avoiding `const`.

[18.4] What does "`const Fred* p`" mean?

FAQ: A pointer to a `const Fred` object. The object can't be changed, for example, you can't call methods not qualified as `const`.

FQA: Right. But don't count on it when you debug code. `const` can be cast away in a snap (pretty much like everything else in C++), and there's the `mutable` keyword for creating members that can be modified by methods qualified as `const`. And of course someone can have *another*, non-const pointer to the same `Fred` object. *And* the `const`-qualified methods may modify data pointed by its member pointers and references or by pointers kept in things pointed by its member pointers, etc.

The pointer aliasing issue is one reason that the compiler *can't* really optimize code because it sees a pointer declared as `const`. And when it *can* figure out there are no aliases, it *doesn't need* your const declarations to help it. You can explain the difference between data flow analysis and type qualification to the next pseudo-performance-aware person who advocates declaring every local integer as `const`. See also a correct FAQ answer to this question below.

[18.5] What's the difference between "`const Fred* p`", "`Fred* const p`" and "`const Fred* const p`"?

FAQ: In the first example, the object is immutable. In the second example, the pointer is immutable. In the third example, both are immutable.

FQA: Um, right. Remember: `const` makes your programs readable.

[18.6] What does "`const Fred& x`" mean?

FAQ: It's a reference to an immutable Fred object.

FQA: Which is similar to a pointer to an immutable Fred object. However, the FAQ holds the "references are NOT pointers" religion (specifically, it belongs to the "pointers are evil" faction), so it dutifully replicates the explanation given in the answer about the pointer case, replacing "pointer" with "reference".

[18.7] Does "`Fred& const x`" make any sense?

FAQ: No. It says that you can change the object, but not the reference. But you can never change a reference anyway, it will always refer to a single object.

Don't write such declarations, it confuses people.

FQA: So why does this compile? Wait, I don't really want to know. Whether it's because they didn't bother to disallow it, or because some special case of template instantiation (like when you add a `const` to a parameter type which is in fact a reference) would fail or anything like that - that's just another lame excuse as far as a language user is concerned.

[18.8] What does "`Fred const& x`" mean?

FAQ: It's equivalent to "`const Fred& x`". The question is - which form should you use? Nobody can answer this for *your* organization without understanding *your* needs. The are lots of business scenarios, some producing the need for one form, others for the other. The discussion takes a full screen of text.

FQA: Yawn. Another stupid duplication in C++.

Come *on*. What "business scenarios" can "produce needs" for a form of a `const` declaration? If your organization employs people who can't memorize both forms or check what a declaration means, you have to admit they will drown in C++ even if you somehow know for sure that in general they can program. C++ has *megatons* of syntax.

[18.9] What does "`Fred const* x`" mean?

FAQ: The FAQ replicates the previous answer, replacing "reference" with "pointer". Probably the same religion thing again.

FQA: Yawn. Another stupid duplication in the C++ FAQ.

[18.10] What is a "`const` member function"?

FAQ: It's declared by appending `const` to the prototype of a class member function. Only this kind of methods may be called when you have a `const` object. Errors are caught at compile time, without any speed or space penalty.

FQA: As discussed above, this breaks into little pieces when your class is similar to a pointer in the sense that a user can change both the state of your object and use your object to change some other state. As to the speed and space penalty, you may check the symbol table of your program if you want to know how many functions were generated twice from a single template because of `const` and non-`const` template parameters. Then you can count the functions having identical code except for extra `const` qualifiers in some versions, but not others.

[18.11] What's the relationship between a return-by-reference and a `const` member function?

FAQ: `const` member functions returning references to class members should use a `const` reference. Many times the compiler will catch you when you violate this rule. Sometimes it won't. You have to think.

FQA: *Of course* you have to think about this complete and utter nonsense! Somebody has to, and the C++ designers didn't.

As usual with `const`, the compiler becomes dumb when levels of indirection appear. For instance, if you keep a member reference, the thing it points to is not considered part of the object. It's up to you to decide whether you want to return this reference as `const` or non-`const` from your `const` member accessor. Both choices may eventually lead to `const_cast`.

[18.12] What's the deal with "`const`-overloading"?

FAQ: You can have two member functions with a single difference in the prototype - the trailing `const`. The compiler will select the function based on whether the `this` argument is `const` or not. For instance, a class with a subscript operator may have two versions: one returning mutable objects and one returning `const` objects - in the latter version, `this` is also qualified as `const`.

FQA: *Please try to avoid this feature*. In particular, get and set functions having the same name (`const`-overloaded) are not a very good idea.

Having to replicate the code of a subscript operator just to add two `const` qualifiers - for the return value and the `this` argument - is yet another example of `const` forcing you to write your program twice. Well, most likely there are more problems where this one came from. A class with a subscript operator is a "smart array", probably not unlike `std::vector`, so you'll end up with much more replicated code - an `iterator` and a `const_iterator` or some such.

[18.13] What do I do if I want a `const` member function to make an "invisible" change to a data member?

FAQ: There are legitimate use cases for this - that would be when a user doesn't see the change using the interface of the class. For example, a set class may cache the last look-up to possibly speed up the next look-up.

Use the `mutable` keyword when you declare the members you want to change this way. If your compiler doesn't have `mutable`, use `const_cast<Set*>(this)` or the like. However, try to avoid this because it can lead to undefined behavior if the object was originally declared as `const`.

FQA: Most compilers probably support `mutable` today, but you may still need `const_cast` because the tweaks to the type system supporting `const` break in many cases. Which may be a problem combined with the fact that `const_cast` is not designed to work with objects declared as `const` (as opposed to those declared non-constant but passed to a function by `const` reference or pointer).

There are numerous reasons making `const_cast` safer in practice than in theory. People rarely declare objects as `const`. Compiler writers are unlikely to add special cases to their compiler to support `const` objects of C++ classes because it's hard work that is unlikely to pay off. For example, allocating a global `const` C-style structure with an aggregate initializer in read-only memory is easy. Doing the same for a C++ class with a constructor is hard because the constructor must be able to write to the object upon initialization. So you'd have to use writable memory you later make readable, which is not typically supported by object file formats and operating systems. Or you could translate C++ code to a C-style aggregate initializer, spending lots of effort to only handle the cases when the compiler can inline the code of the constructors.

However, there is no simple rule making it "very close to impossible" for a compiler writer to use the opportunity provided by the standard and implement `const` objects differently from other objects. In the case of optimizing the dereferencing of `const` pointers, such a rule does exist (there might be other, non-`const` pointers to the same location). But in the case of objects declared `const`, there *should* be no such pointers.

The fact that the compiler is allowed to work under this assumption could be great if it were more likely to actually yield performance benefits, and if `const` worked to an extent making the use of `const_cast` unnecessary. The current state of affairs just creates another source of uncertainty for the programmer.

[18.14] Does `const_cast` mean lost optimization opportunities?

FAQ: Theoretically, yes. In practice, no. If a compiler knows that a value used in a piece of code is not modified, it can optimize the access to that value, for example, fetch it to a machine register. However, the compiler can't be sure that a value pointed by a `const` pointer is not modified because of the aliasing problem: there can be other pointers to the same object, not necessarily constant. Proving the opposite is almost always impossible.

And when the compiler can't prove it, it can't speed up the access to the value. For example, if it uses the value it fetched to a register after someone modified it in the original memory location, the compiler changes the meaning of the program because it uses an outdated value.

FQA: Three cheers to the FAQ! No, really, this time the answer describes the actual state of affairs. You understand why I'm so deeply touched if you met some of the many C++ users who give a different answer to this question, and know their ways to reason about performance.

I think the answer is "no" in theory, too, because basically the compiler has to figure out which parts of the code modify the data, and if it knows the data won't get modified while this piece of code is running, it can use this fact, and otherwise it can't, so what difference do your `const`-qualifications make? But let's say I'm not really sure, and anyway, it's not the time to argue about the exact phrasing when once in a lifetime a realistic statement appears about the performance of C++ code.

The thing that is worth noting is that aliasing problems impede the performance of "generic" libraries, not just compilers. For example, consider `vector<T>::push_back()`. The vector will try to append the object to its storage. If there's no free space left, it will allocate a larger chunk, copy the old objects, and then append the new one. The new object is thus copied once (from wherever it was into the vector storage), right?

But what if the object is itself located in the old chunk, and `vector` tries to be efficient and use `realloc`, which may or may not free the old memory? The object may be wiped out by `realloc`. Very good, then, `vector` creates *another* copy. Hop 1 - from the parameter to a temporary, hop 2 - from a temporary to the vector storage and everyone is happy - after a certain amount of clock cycles.

"Theoretically, yes. In practice, no." Quite a nice summary of C++.

[18.15] Why does the compiler allow me to change an `int` after I've pointed at it with a `const int*`?

FAQ: Because when you point to something with a `const` pointer, this only means that you can't use that pointer to change the object. It doesn't mean the object can't be changed at all, because it can be accessible in other ways, not only through this pointer.

FQA: Exactly; it's related to the previous FAQ. Surprisingly for C++, it even makes sense. After all, when the object is *declared*, it seems like the right place to say what can be done with it. But if anyone can take an existing object and point to it and thus redefine what can be done with it, it's just weird, and can't really work, because it could be done from many places in incompatible ways.

[18.16] Does "`const Fred* p`" mean that `*p` can't change?

FAQ: No, for example, it could change through a non-constant `Fred* q` pointing to the same object.

FQA: This question is just like the previous one.

[18.17] Why am I getting an error converting a `Foo**` to `const Foo**`?

FAQ: It would be "invalid and dangerous"! Suppose `Foo**` points to an array of pointers, which can be used to modify the pointed objects. Suppose C++ would allow to pass this array to a function which expects `const Foo**` - an array of pointers which *can't* be used to modify the pointed objects.

Now suppose that this function fills the array with a bunch of pointers to *constant* objects. This looks perfectly good in that context, because that's what the function was passed - an array of pointers to constant objects. But what we've got now is an array of pointers which can be used to modify those `const` objects, because the declaration of the array does allow such modifications!

It's a good thing we get a compile time error instead. Don't use casts to work around this!

FQA: Wonderful. But why are we discussing evil built-in pointers anyway? Let's talk about the smart C++ array classes. Why can't I pass `std::vector<T*>&` to a function expecting `const std::vector<const T*>&`? The function clearly says that it's not going to modify *anything*: neither the vector nor the pointed objects. Why can't I pass my mutable vector of mutable objects to a function that promises not to modify either of those?

Inheritance -- basics

This section outlines the C++ "support" for the OO concept of inheritance. Do not confuse C++ and OO.

[19.1] Is inheritance important to C++?

FAQ: Sure. That's the new thing OO adds on top of the more basic notion of abstract data types.

FQA: It is, but for a slightly different reason. C++ uses inheritance to implement run-time polymorphism - allow to use two objects of different classes in the same way. In order for this to work, the classes (very roughly) must be inherited from a common base class.

It is also possible to implement polymorphism without inheritance. The system may simply allow you to call the methods of an object without knowing its type, and the code will work if and only if the object actually has the methods with the right name and arguments. Such systems may still support inheritance, but it's less central for them.

OO systems encouraging polymorphism of the former kind are said to be "statically typed" in the sense that you use objects through explicitly defined interfaces which are visible at compile time. This approach is well-established and has many pros and cons (so does the other approach). The C++ choice to use this approach is "legitimate" in the sense that it can yield a relatively usable & consistent result.

However, C++ inheritance is designed to support many other things, leading to a less usable and less consistent result. In particular, inheritance complicates the already complicated name look-up in many ways unique to C++, and multiple inheritance is quite a mess. On the other hand, many handy features are left out, for example, there are no multimethods or dynamic type information allowing to list the interfaces supported by a class (the latter is a special case of lacking reflection).

[19.2] When would I use inheritance?

FAQ: As a tool for specifying the behavior of a system. People think in two dimensions: "is a" and "has a". For instance, a cat "is a" mammal and "has a" tail. Aggregation of data into abstract data types supports the "has a" dimension. Inheritance adds support for the "is a" dimension.

FQA: *When appropriate*. The FAQ itself does a pretty good job listing situations when something "is a" something else on an intuitive level, but using inheritance is still wrong. Inheritance is a formal concept, and formal concepts don't map directly to English words.

Nobody knows the number of dimensions people use to think yet. For example, imagine a black cat. Does it "have a" black color or "is a" black object? What if it's under a table - does it "have a" table above it or "is a" thing under a table? Nobody knows what "thinking" is, and natural language can not be reduced to a couple of dimensions. It "has" an unknown number of dimensions. It "is a" thing with an unknown number of dimensions. Whatever "dimension" means.

A programming language is a tool people use to *program* machines. Even if we knew how thinking works, and we could build a "thinking machine", we'd still need programming languages for *the other* machines. The ones that are supposed to do what they are told, quickly, reliably and cheaply, as opposed to those supposed to be creative thinkers. And a programming language, which must be precise, can not map directly to a natural language, which is not.

To use inheritance or any other formal device, you have to understand exactly how it works. Luckily, in most programming languages it's reasonably easy. And with C++, you can at least try to restrict yourself to the subset you do understand.

[19.3] How do you express inheritance in C++?

FAQ: By using the `: public` syntax, as in `class Car : public Vehicle { ... };`. `private` and `protected` inheritance are different.

FQA: Yeah, very different: replace the `public` keyword with the `private` or `protected` keyword. The FAQ probably means that they are different because they are less related to OO semantically.

You may wonder what the colon syntax has to do with the concept of inheritance. Well, the connection is obvious: there's no inheritance in C, and adding keywords to C++ reduces the "compatibility" with C, because the grammars of these languages prevent you from using keywords as identifiers. Once again, punctation comes to the rescue. The other technique for retro-fitting features into the grammar is overloading keywords (consider `static`).

To be fair, many languages have this problem with their grammar, and it's not a very big deal.

[19.4] Is it OK to convert a pointer from a derived class to its base class?

FAQ: For public inheritance - yes. By definition, an object of a derived class is an object of the base class.

FQA: Moreover, if you don't think someone is going to do this, or you know it's not going to work as expected, don't use inheritance. The whole point of inheritance is to allow exactly that: to have code that works with objects of a base class and in fact can be passed objects of arbitrary derived classes. If making this possible is not what you want, inheritance doesn't really help you and will confuse the users of your classes.

Of course public inheritance is also used for mind numbing template metaprogramming tricks, like having a template derive from its own instantiation recursively, or from a template argument. The judgment of whether this is a reasonable usage of a language feature is left to the reader.

[19.5] What's the difference between `public`, `private`, and `protected`?

FAQ: `private` is for class members and friends. `protected` is also for derived classes and their friends. `public` is for everybody.

FQA: Exactly. The only part having to do with inheritance is `protected`, and it's useful relatively rarely.

[19.6] Why can't my derived class access `private` things from my base class?

FAQ: It protects you from changes to the base class by not letting you rely on its implementation details.

FQA: However, it doesn't protect you from recompilation of your derived classes when changes are made to the base class (access control keywords are little more than comments in C++). Keep that in mind when you design interfaces, especially those which are supposed to be used outside of your team. If you want to supply C++ interfaces, which is pretty daring by itself, not exposing any base classes except pure abstract ones (those having no data members) may be a good idea.

[19.7] How can I protect derived classes from breaking when I change the internal parts of the base class?

FAQ: Your class has two interfaces: `public` and `protected`. To protect your users, don't expose data members. Similarly, you can protect the derived classes by not having `protected` data members; provide `protected inline` accessors instead.

FQA: Dear Design Guru! Listen carefully, and try to understand.

One. Writing get and set function for every member takes time, and so does reading them. This time can instead be used to get something done.

Two. If you can get and set a member, it's pretty close to being public. Any non-trivial representation change becomes impossible since the ability to set this particular member is now a part of the contract.

Three. Having properties in the language - things that look like members but are in fact accessors - doesn't hurt. Many languages have it, allowing to use plain members and in the 0.1% of the cases when your class becomes very popular *and* you want to change it *and* you can do it without breaking the contract, you can make the member a property. There's no reason for not having this in C++ that's even remotely interesting to a language user.

If you know C++ programmers who are obsessed with useless wrapper code *and* performance they never measure, the `protected inline` part is kinda funny.

[19.8] I've been told to never use protected data, and instead to always use private data with protected access functions. Is that a good rule?

FAQ: No, no, no - "always" rules don't work in the real world! You don't have time to make life easy for everyone. Spend your time making it easy for the important people. People are not equal. The important people are called "customers".

FQA: Thanks, that's just what I needed! You know, I was going to make life easier for *everyone* by writing hordes of get and set functions. But now I can *prioritize* and only make life easier for the important people.

Please your customers by writing 56% more get and set functions than your competitor using the newest Visual Refactoring Tool today!

Actually, the FAQ does mention that you'll probably need more than get and set functions to really "protect the derived classes from changes". But the basic assumption that lots of layers make life easy when you have time to create all those layers is still ridiculous. There are teams out there that actually *have* lots of time to do a real world job, and turn a 500-line simple, working, fast prototype program into 30K-line incomprehensible, broken, slow "product" because they think that they are dealing with an *important* task, so now is the time to write piles of wrapper code.

[19.9] Okay, so exactly how should I decide whether to build a "protected interface"?

FAQ: There are many useful guidelines, for example: grow up, don't do things that can jeopardize your schedules, and only invest time in the things which will ultimately pay off.

FQA: If you find this advice useful, here's more: don't be an idiot, be lucky, and avoid pushing sharp objects into your body.

It's rather annoying that the C++ FAQ *dares* to tell people to be "practical". Why don't you add a third useful built-in type to your language (we already have integers and floats), say, a string or a dictionary, and then talk about the difference between theory and the real world?

Inheritance -- virtual functions

This page is about `virtual` functions - a rare example of a C++ feature which is neither a part of C nor completely self-defeating.

[20.1] What is a "`virtual` member function"?

FAQ: The most important thing about C++ if you look from an OO perspective.

With `virtual` functions, derived classes can provide new implementations of functions from their base classes. When someone calls a `virtual` function of an object of the derived class, this new implementation is called, even if the caller uses a pointer to the base class, and doesn't even know about the particular derived class.

The derived class can completely "override" the implementation or "augment" it (by explicitly calling the base class implementation in addition to the new things it does).

FQA: If you are new to OO, using C++ as your first example is not a very good idea. If you still want to start learning about OO here, and you feel you didn't really understand the very generic statements above, here's an example attempting to show why all this is actually *useful*. Note: this FQA is unlikely to be the best OO tutorial on the web.

Suppose you have a program which plays movies in different formats. No matter what the format is, you want to have the same interface (a window with a menu), options (resize the window, control the color balance...), etc. Now, you can define a class `Movie` with `virtual` functions like `nextFrame(&pixels,&width,&height)`. Each format is implemented in a derived class like `MPEGMovie`, `DivXMovie`... The code implementing the interface, the options, etc. can then work with `Movie` objects, calling the right `nextFrame` function without having stuff like `if(format==MPEG) { MPEG_nextFrame(...); } else if ...` all over the place. The format-independent code is easier to read, and much easier to change when you want to add a new format, in fact as easy as it gets: you don't have to change anything.

If this last sentence made you laugh, you've probably seen how representation-specific a supposedly "generic" interface turned out to be once someone actually tried to create a second or a third implementation. You can laugh silently, or you can do it out loud, to soon find yourself surrounded by newbies cluttering their code with idiotic conditions ("the wizard said that nothing is really generic anyway"). So let's forget the whole "pseudo-generic" issue for the moment and focus on the C++ flavor of polymorphism instead (to those who didn't know: hordes of languages have something like `virtual`, and normally it's something better).

`virtual` functions have their problems. The keyword itself is obscure, the `=0;` notation is even more so, you can't look at a derived class and say which functions are `virtual` (the keyword is optional), and overloading makes it even harder to see when a function actually overrides a base class implementation (forget a `const` in the prototype and it becomes an unrelated `virtual` function with the same name). Seasoned OO pros might point out the lack of multimethods and invariants/contracts checking, and have other complaints that I no longer remember, but which seemed valid when I did. Oh, and there's the poor RTTI and the non-existing reflection. It goes on and on.

However, compared to other C++ features, `virtual` functions are excellent. They make a very useful thing easy. You get to type less compared to C, where you would have to create a "vtable" struct and then do things like `pThis->vptr->f((Base*)pThis, args)` instead of the C++ `p->f(args)`. And this brevity does *not* come at the price C++ makes you to pay so promptly - there's no compile time overhead.

Therefore, if you are a practitioner using C++, ignoring the "performance-oriented" brain-washing ("arrays of pointers to objects with virtual methods are soooo slow, arrays of structs with inline functions are soooo fast") and using the single C++ feature that has a potential of doing less harm than good is very frequently the right thing to do. Of course not using C++ for code characterized by complicated structure and dynamic behavior is an even better idea.

[20.2] How can C++ achieve dynamic binding yet also static typing?

FAQ: *Static typing* means that when a function (`virtual` or other) is called, the compiler checks that the function call is valid according to the statically defined interfaces.

Dynamic binding means that the code generated from the statically checked function calls may actually call many different implementations, and figures out the one to call at run time.

Basically C++ allows many things to be done upon a `virtual` function call, as long as these things follow a specified protocol - the base class definition.

FQA: It's important to keep in mind that the compiler can only check whether an implementation follows a protocol to a certain extent. It is pretty easy to create a derived class which looks legitimate (it has an `insert` and a `find` method as specified in the base class), but in fact it is not (because `insert` doesn't really insert anything to anywhere, and `find` doesn't find anything). This will break the code using the derived class via pointers to base class objects (this code will `insert` something and then won't `find` it and will crash). That's why the FAQ has a whole section on proper inheritance.

The (natural) fact that you can't find all errors with static typing wouldn't be that bad if C++ didn't crash as hard as it does (for instance, by doing run time boundary checking), and/or would support dynamic contract checking, and/or had a clear type system that would make specifying interfaces less of a pain (currently you have a zillion troubles ranging from no built-in string type to the endless kinds of type relationships and conversion rules).

Dynamic binding is a critical feature to build extensible software, and all general-purpose programming languages have some form of it (in particular, C has function pointers). Unlike dynamic binding, static typing is not strictly necessary, but it has many benefits (the code may be easier for people to read and for compilers to validate and optimize due to the information specified in the types) as well as many drawbacks (some things may be hard to model in a static type system, frequently there's more code to read and write, etc.). At the language level, C++ doesn't support dynamic *typing* (as opposed to "binding") at all, and its static type system is one of the worst.

Some C++ programmers think (or feel) that their compiler does an incredible service of finding virtually all of their bugs. Their hidden motives include the need to rationalize the amazingly long build cycles, and to avoid writing test programs (more verbose, ugly, boring C++ code - who wants to do that?!). You'll do yourself a favor by not catching their thinking habits.

[20.3] What's the difference between how `virtual` and non-`virtual` member functions are called?

FAQ: non-`virtual` functions are resolved at compile time. `virtual` functions are resolved at run time.

The compiler must generate some code to do the run-time resolution. This code must be able to look at an object and find the version of the function defined by the class of the object. This is usually done by creating a table of virtual functions for each class having them (the "vtable"). All objects of the class have an implicitly generated member pointer (the "vptr"), initialized to point to the class vtable by all constructors.

To implement a virtual function call, the compiler generates code similar to that it would produce from the C expression `(*p->vptr->func_ptr)`. That is, it dereferences the object pointer to fetch the vptr, then fetches the function pointer from a fixed offset, then calls the function through the pointer.

The exact cost depends on complicated stuff like page faults, multiple inheritance, etc.

FQA: In practice, the cost is almost never a big deal unless you call `virtual` functions in your "innermost loops" (the code processing the most data). I'm not saying the cost should be ignored - a good-looking generic design centered around virtual calls in innermost loops is invalid for most practical purposes. I'm just saying that you don't have to think about things like page faults to figure out if you can afford `virtual` in this piece of code. The C++ implementation of polymorphism is pretty efficient.

One kind of price you pay for this efficiency is the instability of your interfaces at the binary level. You *can* add new derived classes to your system without recompiling old derived classes. But you *can't* add a virtual function to a base class without such recompilation. This rules out straight-forward usage of virtual functions in many situations (if you think recompilation has "zero cost", try to get the vendors of your desktop software to recompile it so it runs on a different processor architecture).

Many OO languages avoid these problems. One way to do it is just-in-time compilation - instead of generating code fetching the function pointer from a fixed vtable offset, wait until you see all the updated base class definitions at program load time and then generate the offsets. Another option is to use less efficient, but more flexible look-up mechanisms, such as hash tables - this is typically coupled with the lack of static typing, which has many benefits and drawbacks. Both techniques avoid a huge amount of real-life problems with binary interface stability.

If you can't switch to a different OO language with better OO support (think about it - the interface stability is just one aspect, you'll probably also get garbage collection, faster build cycles, and a ton of other useful things), you can work around these problems in many ugly ways. For example, you can have a single `virtual` function getting a string or a `void*` telling it what to do (a famous example of achieving binary-level stability this way is the `ioctl` system call of Unix-like systems). Another approach is to add new classes for the new functions, and have the derived classes implement many different abstract base classes (the COM `QueryInterface` function works this way). These things are not pretty, but they are better than nothing, which is what `virtual` functions deliver when you need stable interfaces.

Don't be ashamed of yourself if you reach the conclusion that you have to do this. If "performance-aware" people squeak something about it, try to get them busy with something else, like implementing `std::valarray` to actually compile to fast code using SIMD instructions of the host processor, without creating temporary memory objects. This should give you a couple of centuries long break.

[20.4] What happens in the hardware when I call a `virtual` function? How many layers of indirection are there? How much overhead is there?

FAQ: There's a lot of code listings explaining the previous answer in detail.

FQA: By "hardware", you probably meant "binary-level software" - for each popular binary instruction encoding there are lots of variants of processors implementing it, and the processors are integrated into many different systems with all kinds of memory architectures. Even when you write *assembly code*, you can't tell exactly how it's going to be executed, so of course you can't with a higher-level language which can be compiled to assembly in many different ways.

As to levels of indirections - typically there are two, one to fetch the vptr from the object and another one to fetch the function from the vtable. One other approach could be to save a level of indirection by keeping the vtable inside the object "by value" instead of "by pointer" - but that makes the objects larger.

As the proponents of `virtual` methods correctly point out, some overhead is inevitable in the situations where you use a `virtual` function, because you don't know what function to call at compile time. Of course different ways to implement the decision making can have slightly different performance characteristics on a given platform. If you care about such tiny differences though, you probably have to fix it at a higher level (such as moving the decision outside of the innermost loop at the cost of possibly replicating some of the common code). Fiddling with the low level - implementing the decision - is lots of boring work (rewrite, measure, rewrite, measure...) with little gain.

[20.5] How can a member function in my derived class call the same function from its base class?

FAQ: For some a reason, a pretty low-level discussion follows. Name mangling, "call-by-name" vs "call-by-slot-number", code listings with double underscores...

FQA: In a `class Derived`, in a function `Derived::f()`, you can type `Base::f();` to call the `f` implementation from your base class `Base`. The compiler will ignore the actual type of your object (at the low level - vtable and all that), and call `Base::f` just the way non-`virtual` functions are normally called.

I think you can do that in every OO language. It's pretty natural.

[20.6] I have a heterogeneous list of objects, and my code needs to do class-specific things to the objects. Seems like this ought to use dynamic binding but I can't figure it out. What should I do?

FAQ: "It's surprisingly easy", says the FAQ. This statement is followed by a surprisingly long answer.

FQA: I don't know how "surprising" this is - that's the whole (the one and the only) point of polymorphism: to do class-specific things to objects, without thinking how heterogeneous they are.

For this to work, have the classes of those objects derive from a common base class (not extremely "easy" unless you planned ahead...). Then you can run over the elements of `std::list<Base*>`, and call the virtual methods which do the class-specific things without thinking about how many different classes the objects actually belong to, etc. For example:

```
for(std::list<Shape*>::const_iterator p=shapes.begin(), e=shapes.end(); p!=e; ++p) {
  (*p)->draw(window);
}
```

Off-topic: the STL way of saying `foreach p in shapes` is pretty ugly, isn't it?

[20.7] When should my destructor be `virtual`?

FAQ: The rule of thumb is - when you have a `virtual` function. Strictly speaking, you need it when you want someone to be able to derive classes from your class, create objects with `new`, and `delete` them via pointers to a base class.

FQA: The situations when the rule of thumb is not good enough were not reported on our planet. Use this rule of thumb, if only to suppress compiler warnings. Too bad the C++ compiler doesn't use the rule silently itself - it could simply make the destructor `virtual` in these cases.

[20.8] What is a "`virtual` constructor"?

FAQ: It's an idiom allowing you to have a pointer to a base class and use it to create objects of the right derived class. You can implement it by providing `virtual` functions like these:

```
virtual Base* create() const;
virtual Base* copy() const;
```

You implement them like this:

```
Derived* Derived::create() const { return new Derived; }
Derived* Derived::copy() const { return new Derived(*this); }
```

Note that we return `Derived*`, not `Base*` - it's called "Covariant Return Types". Some compilers have it, some don't.

FQA: Other languages have built-in support for these things. This is interesting because of the *other* things they make possible, like serialization (without writing special code for each class). Roughly, this is called "reflection": you can find all methods of a class at run-time, including those that are not normally called as `virtual`, such as constructors (which can't be `virtual` - you have to create an object *before* you can dispatch function calls based on its type). Another option is to (recursively) get the list of members of a class and create/copy them. It's hard to imagine how useful this is if most of your programming experience comes from working with C++.

Covariant return types are a nice joke (for an admittedly narrow audience though). C++ lets you tighten the specification of return values - which is perfectly legitimate: our base class said we should return a `Base*`, and we surely implement the contract if we always return a `Derived*`, which is in particular a `Base*`. *But* C++ doesn't let you *loosen* the specification of arguments, which can be shown to be legitimate for symmetrical reasons.

Why? Because C++ has overloading, so when you declare a `virtual` function which looks just like a function from your base class, but has any kind of changes to argument types, C++ thinks you create a new unrelated `virtual` function rather than override the one from the base class. Since C++ has no overloading based on the function return type, there's no symmetrical problem.

C++ has lots of kinds of static typing, but little consistency between them. To be fair, other languages have similar interactions between overloading and dynamic binding, and some probably copied them from C++. However, other languages rarely encourage design which depends on the availability of overloading to work (like STL), and/or is based on the microscopic details of overload resolution mechanisms (like some of the boost libraries).

Inheritance -- proper inheritance and substitutability

This section is about using inheritance such that the code really works, not just compiles. Unlike most "OO"-related sections in the FAQ, much of the material is applicable to decent OO systems and not only to C++.

[21.1] Should I hide member functions that were public in my base class?

FAQ: No, no, don't even think about it, *don't do that*, no. Your desire is probably the result of "muddy thinking".

FQA: With all due respect, it is your precious programming language that probably is the result of "muddy thinking". The question talks about overriding base class functions in the `private` section of your derived class. This is trivially and reliably detectable at compile time. If you get so excited about how wrong it is, *why does it compile*?

Answer: in C++, random things compile and other random things don't. The language definition is sloppy. What's that? You think the compiler writers made their own job easy by making yours hard? No, C++ is probably the hardest language to compile among those popular today. C++ is *pointlessly* sloppy.

The reason the FAQ gets so excited will become clear in the next answers. Basically, when your derived class overrides a function as `private`, you violate the substitutability principle: it is no longer true that an object of a derived class fully supports the interface of the base class. However, technically the functions from the base class are still accessible, because you can cast a pointer to a derived class object to the base class and call the function through the vtable (pretty *muddy*, isn't it?).

People define overridden `virtual` functions as `private` to convey the message that objects of the derived class should never be used directly, and the purpose of the class is to interact with a framework which works with objects through base class pointers. While the FAQ gets overly hysterical about this practice, the polarity of its answer ("no") is probably right.

[21.2] Converting `Derived* -> Base*` works OK; why doesn't `Derived** -> Base**` work?

FAQ: Because it shouldn't. Let's pretend it does work and see what happens.

Suppose you have a `Dog* d`. You pass it to a function `void f(Pet** p)` with `f(&d)` - which should be OK, since `Dog` is derived from `Pet`. The function does this: `*p = new Cat;` - perfectly legitimate, since `Cat` is derived from `Pet`, too. But now we have a `Dog*` pointing to a `Cat` object. So `d->bark()` will crash the program, or misbehave more severely, since a `Cat` may have a virtual function `scratchFurniture` at that slot of the vtable.

Actually, the FAQ uses a scarier example, which launches nuclear missiles as the result of the mistake. IMHO, nothing can beat the following classic in this department:

```
if(status = UNDER_ATTACK) {
  launch_nuclear_missiles();
}
```

Best Industry Practice: use peer reviews to increase the quality of your nuclear missiles launching code.

FQA: Yep, levels of indirection and static typing interact in non-obvious ways. This is another incarnation of the problem making it impossible to cast `T**` to `const T**`. Basically, *a T* is always a S** doesn't mean *a T** is always a S***.

The problem is that there are many cases where you *know* that you are doing something legitimate, but the compiler doesn't. For example, you know that it was *you* who filled this vector of Pets with a bunch of Dogs. You *couldn't* use a vector of Dogs because you wanted to pass it to a function working with a vector of Pets. And as we've just seen, the compiler wouldn't let you pass a vector of Dogs to a function expecting a vector of Pets, and for a good reason. So you ended up with a vector of Pets filled with Dogs. And now you want to fetch a Dog from the vector - but the elements are typed as Pets, so you have to use a cast. It wouldn't be that bad if these cases wouldn't cause many people to develop a habit of aggressive casting to have the compiler shut up, and/or C++ would catch illegal cast operations at run time.

Moral: static typing (having the compiler validate the code according to a set of rules specifying properties of types and their relationships) is hard. A static type system will get in your way. And it only partially compensates you by "validating the interfaces", because only some of interface specification can be modeled statically, as we'll see below. In particular, consider our example where you had to stuff your Dog objects into a vector of Pet pointers, all because the compiler *insisted* on the looser typing. Now the compiler won't prevent someone else from adding a Cat pointer to that vector, and then your code fetching a Pet* from the vector and casting it to a Dog* will misbehave.

I'm not saying that static typing is "bad", but if you think that *dynamic* typing is bad, you are very lucky - you're just one step away from a quite noticeable increase in your productivity. Pick a dynamically typed language and give it a try.

[21.3] Is a parking-lot-of-`Car` a kind-of parking-lot-of-`Vehicle`?

FAQ: No, because a `Plane` is one kind of `Vehicle`, and you don't want someone to park it at a cars' parking lot.

FQA: In English, apparently the answer is yes. In OO, the answer is no. In natural language, there's no strict definition of "kind-of" (or anything else, for that matter). OO systems are formal, and they have a precise definition for "kind-of": *B is a kind of A if you can do to a B object whatever you can do to A, and it will work correctly* (not just compile).

Programming languages are not natural languages. In particular, the good programming languages don't try to look "natural" when such attempts make it hard to understand the formal, precise and dumb stuff the machine actually does. If you ever wondered what on Earth the C++ expression `a->b` does (when `a` is an object of a smart pointer template class with 7 parameters), you know what I mean.

[21.4] Is an array of `Derived` a kind-of array of `Base`?

FAQ: No. Think of the array as an implementation of a parking lot, and you'll see that the answer follows from the previous FAQ.

FQA: Note that the ability of the compiler to figure out whether something is a kind-of something else is limited. In particular, it seems to work better with types related by inheritance (base and derived classes) than with types related by qualifiers (`const` and non-`const`) or by the way they are instantiated from the same templates. For example, a `vector<T*>` is apparently a kind-of `const vector<const T*>`, because there's nothing you can do with an all-`const` vector you couldn't do with an all-non-`const` vector. But the compiler doesn't know that.

One way around this is "duck typing" - don't bother to specify the relationships between the types, just pass objects to functions, which will work if the object can do whatever they ask it to do, and raise a run time error otherwise. "If it walks like a duck then it is a duck" and all that - you don't have to define a `Duck` interface all ducks should follow, just get an object and call methods such as `walkLikeADuck`. C++ doesn't have duck typing because it would require the compiler to rely on non-trivial and not-so-lightweight run time mechanisms, which kind of goes against the "spirit" of C++ (not that the run time mechanisms used to implement exceptions are trivial, mind you).

One could claim that duck typing is incompatible with the "spirit of C++" because it involves run-time dispatching, but so do `virtual` functions, which are more efficient but less flexible and much more likely to trigger recompilations - a big deal in many situations. Or one could claim that duck typing is not "the C++ way" because it leaves out the specification of interfaces, but so do templates, which provide "static duck typing" - too bad they are such a pile of toxic waste that the scope of this discussion is too narrow to even briefly describe why. Or one could claim that with duck typing, you can fail at run time because someone provided an object of the wrong type - but nothing prevents someone from simply passing a null pointer to a C++ function that can't handle that and have it crash much harder than any code in a safe dynamic language ever will.

The true reason making duck typing incompatible with The C++ Way is the 95% Is Nothing Axiom. It goes like this: "if something is only useful for 95% of the cases, *and* it doesn't map almost directly to C, it's not worth adding to C". Other examples of the application of this axiom to the design of C++ is the lack of garbage collection, which "only" handles memory (>95% of all "resources"), and "only" in non-real-time applications (>95% of all application code).

The consequences of this axiom wouldn't be that bad if the features C++ *did* add to C were any good.

[21.5] Does array-of-Derived is-not-a-kind-of array-of-Base mean arrays are bad?

FAQ: Yes, arrays are evil. Normally you should use `std::vector` instead of arrays. But if you are an enlightened OO specialist and so is everyone likely to maintain your code, and you fully understand the interaction of "kind-of" and arrays, you may use them.

FQA: Huh? Arrays and vectors are synonyms in the context of the "kind-of" issue. What does the cult advocating the replacement of C features, which have their problems, with new shiny C++ features having much worse problems have to do with proper inheritance?

What's that? Casting arrays is easier than casting vectors? Try this: `(vector<T>*)&vec_of_something_else_than_T`. Seriously, this is one weird question with a strange answer we have here.

[21.6] Is a `Circle` a kind-of an `Ellipse`?

FAQ: Sometimes it is, most frequently it isn't. For example, if an `Ellipse` lets you change the size in a way making it asymmetrical, it's not a `Circle`.

The point is that if you derive a `Circle` from an `Ellipse` and then someone tries to use an `Ellipse*` which really points to a `Circle` object, there's no way to make it work gracefully. Either the calling code will get an error in some form, even though it does something which should be possible to do with an `Ellipse`, or the `Circle` object will obey to the caller and become an invalid circle, breaking some other legitimate piece of code which does expect it to be a valid circle.

FQA: This is just like the parking lot example in the sense that "kind-of" in English means many different things, some of which are incompatible with the precise definition of "kind-of" used in OO. The important point is that the interfaces are protocols and implementations must follow them.

Some people think about inheritance merely as another form of "binding" - having the compiler call a function using new syntax. From this point of view, everything is legitimate as long as the program compiles and does whatever the end user expects. But this way inheritance only makes programming harder (another kind of syntax to decipher). The more restrictive "interfaces as a protocol" approach can make programming easier because when you implement a bunch of protocols correctly, you can extend a program without tweaking its code (for example, add a movie format to a media player). But this only works if you *really* follow the protocol. If you *sort of* do it ("a Circle is a kind-of Ellipse, well, almost - just don't call this function"), the media player will crash.

There are numerous families of examples where natural languages and OO terms are not aligned (which doesn't mean OO is bad - it means it's formal, which is good for computer programming). The "parking lot" represents one family (collections); Circle/Ellipse represent another one (parametric representations). One family of "positive" examples (where inheritance is likely to be proper) is record types (a `CPlusPlusProgrammer` has all the fields of a `Programmer`, plus a couple of new, orthogonal members, such as `headAgainstTheWallBangingFrequency`).

[21.7] Are there other options to the "`Circle` is/isnot kind-of `Ellipse`" dilemma?

FAQ: Well, you need to get rid of *some* of your original claims to get back to consistency. Either `Ellipse` has no `setSize` function which can make a circular `Ellipse` object non-circular, or there's no inheritance which makes it possible to call such a function on a `Circle` object, or you can even choose to live with the fact that some of your `Circle` objects will become non-circular (and have the code working with `Circle` objects deal with it).

Trying to keep all claims and cover up the problem by doing "something reasonable" (like calling `abort` when `setSize` is called with a `Circle` object, or "fixing" its arguments) is not going to solve the problem, because ultimately it breaks the assumptions behind the calling code.

FQA: The FAQ answer is apparently correct and complete. Incidentally, this isn't exclusively about C++, it's about OO in general.

One solution is to have `setSize` return a new `Ellipse` object. This way, `Circle::setSize` will return a `Circle` unless the new size is asymmetrical, in which case it will return an `Ellipse`. One possible benefit is efficiency - circles have less parameters than ellipses, so if you have lots of operations to do with a bunch of objects, you'd rather have all of the objects that can be represented as `Circle` objects actually *be* represented that way, not as redundant `Ellipse` objects.

If you "roll your own OO" (that is, implement inheritance yourself instead of directly relying on language features), you can avoid the creation of a new object and instead dynamically change its type. For example, `setSize` may change the vptr to point to an `Ellipse` vtable when the new size is asymmetrical. This kind of thing is implemented in the POV-Ray ray tracer, written in C.

The fact that you can't do it in a portable way with C++ inheritance probably *doesn't* mean that C++ inheritance is underpowered (surprise!) - you need this kind of thing once in a lifetime, and you must have it very well thought-out to make it really work, and in these rare cases you can go ahead and use function pointers instead of inheritance and implement it. There probably are people that would classify this limitation as a symptom of a deeper problem - having too much logic built into the compiler and too little ways to implement compile time logic in user code - but it's debatable.

[21.8] But I have a Ph.D. in Mathematics, and I'm sure a Circle is a kind of an Ellipse! Does this mean Marshall Cline is stupid? Or that C++ is stupid? Or that OO is stupid?

FAQ: It means a different thing: your intuition is wrong in the sense that it leads you to make wrong decisions about inheritance. The right way to think about "kind-of" is this: B is a kind of A if you can always substitute a B for an A.

FQA: I like how this question is formulated. Shows spirit. In general, the FAQ can be quite entertaining if you're into that sort of thing. If I could legitimately quote the answers instead of summarizing them, I'd sure would.

Which is all nice and dandy, but did you notice the disturbing claim "your intuition is wrong"? Instead of admitting that OO is *not* a natural language, and it *doesn't* have to map directly to a natural language, the FAQ actively tries to persuade you to *change* the way you use natural language words to make your thinking OO-compatible. Next, they'll ship patches you should apply to your DNA, and a sticker saying "Designed for C++ Programming" for your skull.

I think this point is worth discussion because it's representative of the whole notion of "good" in the C++ world. C++ tries to make the program *look* natural. See - we add things with the plus sign, and errors are handled transparently, and resources are managed automatically - that's one very high-level language, and it's efficient, too! But make a single error in your program - and finding it becomes an nightmare. What is *really* being called by this `a+b` expression? What *really* happens upon error? And this object we deallocate here - how do we know nobody is keeping a pointer to it? Because all our pointers are "smart"? But look - here we use a library using bare pointers, and here's one using different smart pointer classes. What is *really* going on here?

The basic rule C++ breaks is this: don't make promises you can't keep. Don't say that inheritance is equivalent to the way people think about "kind-of" - introduce it from the beginning in terms of substitutability. Don't pretend you manage resources "automatically" when in fact it's the responsibility of everyone to follow non-trivial protocols for this to work, and a single error is fatal - make it visible where resources are acquired and released. Or you can *really* manage them automatically - with garbage collection or reference counting or otherwise. But if you refuse to do it, which may be perfectly legitimate at times, *admit it*. Changing your terms is more productive than waiting for everyone to change theirs.

Of course the Circle/Ellipse problem is *not* an example of "making promises that can't be kept". It's the FAQ's claims about OO "capturing the way we think" that are such an example.

[21.9] Perhaps `Ellipse` should inherit from `Circle` then?

FAQ: Probably not. For example, what would the `radius()` accessor do, and how would it be compatible with an assumption that is most likely a part of the `Circle` protocol that you can use `radius()` to compute the `area()`?

FQA: I think it's very easy to see with a slightly different, but a related example. What is more stupid: to claim that a triangle is a rectangle with two identical vertices, or that a rectangle is a triangle with 4 vertices? It probably sounds equally stupid to most people.

The major reason making people who themselves would think these claims are stupid to go ahead and derive `Triangle` from `Rectangle` or vice versa is that *they don't think they are in fact making such claims by implementing such inheritance*.

The idea is this: inheritance is not just yet another kind of syntax. Its purpose is *not* to save a couple of lines of code in the derived class (which you may accomplish by deriving `Triangle` from `Rectangle`). And the compiler *can't* check that your inheritance is correct (this is really hard for C++ aficionados to accept: *the compiler can't check something!*). Inheritance is about writing code that follows a protocol, making it possible to call this code from any function written to work with objects that follow that protocol, and thus *reusing the calling code* (possibly *a lot* of such code - much more than the couple of lines you saved in the derived class).

And if your inheritance does not guarantee substitutability, then the compiler won't be able to catch your error (it's type checking *assumes* that you provide substitutability - that's why it lets you use pointers to derived class objects in contexts expecting base class object pointers). And you'll confuse most people (frequently including yourself), who also expect substitutability, especially since the compiler agrees by letting them pass an `Ellipse` where a `Circle` is required. And if you really don't need substitutability, you don't really need (public) inheritance, either.

[21.10] But my problem doesn't have anything to do with circles and ellipses, so what good is that silly example to me?

FAQ: But you see, *all* examples of improper inheritance are basically equivalent to the `Circle`/`Ellipse` case. Inheritance is bad when a base class provides functionality which a derived class can't provide (in the `Ellipse` case, that's asymmetrical resizing). The problem with inheritance in such cases is that it comes without substitutability, breaking a basic assumption shared by programmers using the classes and the compiler (which automatically allows to use objects of derived classes where base class objects are expected).

FQA: Exactly. People obsessed with compile-time error checking, repeat: the compiler does the static type checking (as in "this object is of class `Derived` - OK, it's a legitimate parameter to function `f(Base&)`") based on assumptions it can not check ("whoever wrote `Derived` made it substitutable for `Base`"). Say it again: *the compiler does the static type checking based on assumptions it can not check.* **The compiler does the static type checking based on assumptions it can not check.**

Translation: the correctness of an interesting program can not be checked at compile time. All programmers are supposed to know it, but some keep forgetting. So is it ultimately better to spend your time on type safety (things like making sure that nobody can cast `vector<T>::iterator` to the underlying `T*`) or writing tests checking that your code behaves correctly at run time? You be the judge.

[21.11] How could "it depend"??!? Aren't terms like "Circle" and "Ellipse" defined mathematically?

FAQ: They are, but the *classes* `Circle` and `Ellipse` have a *different* definition - the C++ code defining the classes. In your program, that's the definition of `Circle` and `Ellipse`, and that's what you have to look at to validate your inheritance. If you keep thinking about the mathematical connotations, let's replace the class names with `Foo` and `Bar` for the moment; that's all the same for the compiler.

Now that we've defined the meaning of `Circle` and `Ellipse`, recall that "inherits" means "is substitutable for" (not "is a" or "is a kind of", which are not precise definitions). With these definitions, you can get the right answer using the previous FAQs.

FQA: Exactly - you can't implement the mathematical notion of "circle" in a programming language, you can only implement a definition (possibly called `Circle`) or a bunch of definitions which model some of the aspects of mathematical circles to a certain extent. And when you reason about the correctness of your program, you have to talk about these definitions, not the original mathematical notion.

Lots of suffering inflicted by the more talented programmers upon themselves originates at the hope to implement "the ultimate something" (for example, "the ultimate circle class" that captures *all* aspects of mathematical circles, so you'd never have to define a circle class again). The *ultimate* search for "the ultimate something" in programming is probably the search for *the ultimate programming language*. Arguably, the C++ language is one result of this search - it tries to meet a huge amount of conflicting requirements, the key ones being "readability, efficiency and generality" of C++ code, as well as pseudo-compatibility with C. The result is a large-scale nightmare, and the moral of the story is simple: design the best tool for everything, and you'll get a tool good for nothing.

On the bright side, it is probably possible to define a good `Circle` class for your program - if you try to make it good *for your program* rather than implement the mathematical notion. And this is why the meaning of `Circle` *depends* on your program.

[21.12] If `SortedList` has *exactly* the same public interface as `List`, is `SortedList` a kind-of `List`?

FAQ: It's quite unlikely. For instance, consider `List::insert`. Is it defined to insert the element to the end of the list? If it is, there's no good way to implement it in `SortedList`, because the insertion to the end will usually make the list unsorted.

The substitutability principle is about the specified behavior, not just function names and parameter types. So "exactly the same public interface" in the syntactic sense is not enough - for proper inheritance, the specified run time behavior must be the same.

FQA: Yep, compile time type checking can not guarantee proper inheritance, it can only operate under the assumption that *you* guaranteed it. That's why some languages come with contract checking: the base class specifies the behavior using input and output constraints computed at run time, and you can have your run time environment automatically evaluate these constraints when methods of derived classes are called.

You can simulate this behavior in C++ by writing lots of code. Namely, the base class can have a public non-virtual `insert` method calling a protected virtual `onInsert` method. The `insert` wrapper can then check whether `onInsert` follows the protocol using a bunch of `assert`s before and after the call to `onInsert`. Since "a lot of code" is most frequently bad by itself (because you waste time writing it and then waste much more time reading it together with other people), the benefits are not necessarily worth the trouble. But run time tests (stand-alone or integrated into a larger system) greatly increase the quality of code, and making run time testing simple and painless pays off, especially compared to work spent on compile time error detection.

Inheritance -- abstract base classes

C++ abstract base classes are not bad at all by C++ standards. However, they are quite poor by OO standards.

[22.1] What's the big deal of separating interface from implementation?

FAQ: Interfaces are the most important thing possessed by a company. Defining interfaces is hard, throwing together an implementation is easy. Designing interfaces takes lots of time and is done by people who are paid lots of money.

You wouldn't want these valuable artifacts clobbered by implementation details, would you?

FQA: If interface is decoupled from implementation, you can have more than one implementation of the same interface. This way, code *using* the interface can work with all those implementations. For example, the same code may be used to render a document to a screen, a printer and a file. Which is *excellent*, and is a *very widely acknowledged and basic fact*.

It's amazing that the FAQ's answer fails to mention this. The FAQ's claim about the relative importance of interfaces and implementations is unbelievable, too. Here's an interface for you:

```
class SearchEngine
{
public:
  // search the documents pointed by URLs in the inputURLs stream, search the query string
  // in them and emit the URLs sorted by relevance to the outputURLs stream (at most maxResults URLs).
  virtual void find(const string& query, istream& inputURLs, ostream& outputURLs, int maxResults) = 0;
};
```

Now, that took about 1 minute. It's your turn to "throw together an implementation", which shouldn't take more than 30 seconds according the FAQ's argument, should it? Go.

If you don't like this example ("there are more complicated interfaces in a real search engine"), think about `matchScore = faceRecognition(suspectImage, testImage)`, `fft(outputArray, inputArray, size)`, `text = ocr(imageWithHandWriting)`, `render(screen,htmlPage)`, `write(file,buffer,size)`... The idea that "implementations are trivial" can only come from working on extremely stupid applications or example programs for text books.

Of course many interfaces should be decoupled from implementations - but that's because they *hide* a *much more complicated* implementation, and/or because they have more than one implementation, and/or because you want to change the implementation without affecting the client code. Surprisingly enough, C++ abstract classes actually allow you to achieve all these things.

Of course they don't fix many hard problems with C++ - there's no real run time type information, for example. And adding a function to an abstract class changes the binary interface, which is frequently unacceptable. But abstract classes are better than nothing, as opposed to the many C++ features which are in fact *worse* than nothing.

[22.2] How do I separate interface from implementation in C++ (like Modula-2)?

FAQ: Use an abstract base class.

FQA: What do you mean "like Modula-2"?

If you want to be able to work with many different implementations selected at run time, abstract base class is the way to go. Alternatively, you can roll your own "abstract base classes" by using structures containing function pointers, emulating "vtables". This results in extra code, but it's more portable (C binary calling conventions on a given processor architecture tend to be shared by more tools than C++ ones).

If you don't need dynamic implementation selection, you can use a header file with forward declarations of types, and have the types defined in the implementation file(s). This way you can change the implementation details without recompiling the calling code (which you can't do with C++ `private` members). In particular, you can build several versions of your program with different implementations of this interface without recompiling the entire program.

[22.3] What is an ABC?

FAQ: An abstract base class.

Logically, it corresponds to an abstract concept, such as `Animal`. Specific animals, like `Rat`, `Skunk` and `Weasel`, correspond to implementation classes derived from `Animal`.

Technically, a class with at least one pure virtual function is abstract. You can't have objects of abstract classes, you can only create objects of classes derived from them that implement all the pure virtual functions.

FQA: ABC is also an especially annoying abbreviation.

Here are a couple of things you can't do with a C++ abstract base class, and which are supported in other OO languages. You can't iterate over the methods of an abstract base class (neither at compile time nor at run time). So you can't easily utilize the decoupling between the interface and the implementation by automatically generating wrapper classes that log function calls or pack the arguments and send them to an object in another process and/or on another machine. You also can't ask a random object whether it implements the given interface (in language terms, whether its class is derived from a given abstract base class).

If the only language you use heavily is C++, it's possible that you don't realize that you might *want* to do these things at all, or even memorized elaborate arguments "proving" that these things *should not* be done. But hey - at least abstract classes don't make anything *worse* than it already was in C. Quite an achievement for a C++ feature.

[22.4] What is a "pure virtual" member function?

FAQ: It's a virtual function which (normally) has no implementation in the base class, and (always) must be implemented in a derived class to make it non-abstract. The syntax is: `virtual result funcname(args) = 0;`.

This declaration makes the class abstract - it is illegal to create objects of this class. A derived class that doesn't implement at least one pure virtual function is still abstract. Only derived classes without any unimplemented pure virtual functions can be instantiated (you can create objects of these classes).

FQA: The syntax is ridiculous. You may wonder what `virtual result funcname(args) = 1999;` means. Well, it means nothing; it doesn't compile.

One possible motivation for this syntax is to save a keyword (such as `pure` or `abstract`). The more new keywords there are, the "less" C++ is compatible with C, or with C++ code written before the new keyword was introduced (well, strictly speaking, either two things are compatible, or they are not, but we'll ignore that for the moment). That's because C and C++ have grammars which disallow to use keywords as identifiers. This problem is shared by the majority of programming languages.

However, C++ does have a huge amount of new keywords. This makes one wonder why the very rarely used `explicit` (not to mention `export`) is a keyword, while the many different uses of the keyword `static` are in fact collapsed into a single keyword instead of using new keywords for the new uses. And why not use the silly, but standard "reserved namespace" (names prefixed with two underscores or a single underscore followed by a capital letter are reserved for the compiler), the way it's (mostly) done in C99? Then we'd have `__explicit` built-in and `explicit` would be a `#define` you'd `#include` from `<shiny_new_useless_cxx_features>` (no trailing `.h`, of course). This way, you give a normally looking keyword to people who need it, and don't break the code of those who don't.

Here's a proposal for the next C++ standard: let's define two keywords, `__0` and `__1`. With a token sequence composed of these two keywords, we can express anything (actually, one keyword is enough, but that's just too verbose). Then we won't ever need any new keywords. As the pure virtual syntax example shows, the readability loss is a small price to pay for the forward, backward and downward compatibility achieved using this approach.

[22.5] How do you define a copy constructor or assignment `operator` for a class that contains a pointer to a (abstract) base class?

FAQ: If you don't want to make a "deep copy" of that pointer, there's nothing special about this case.

If you do want to make a deep copy of the pointer (which you normally should do when the class "owns" it), the abstract base class should have virtual `copy` and `assign` methods you can call from your copy constructor and assignment operator.

FQA: This is a small example of how C++ forces you to write code which could be generated automatically. You can avoid the problem by avoiding the ownership approach to lifetime management (RAII), but figuring out that you need these virtual functions and implementing them is easy for people who understand C++ well.

The trouble is that most people using C++ *don't* understand it very well. C++ has the lethal combination of looking all nice and simple on the surface and having lots of traps installed under the surface, waiting for a victim. The only chance of the victim is to have a very good understanding of the underlying problems of C++. With that understanding, it's easy to see that, um, you have to make a deep copy since you're the "owner", and an object pointed by two owners is going to be wiped out twice by the destructor, and we don't want that, so let's copy the object - costly, but safe, and, um, we can't just copy the object, because we don't know its type, but wait, the object *itself* does know its type, so we need it to help us with the copying, so we need to add these virtual `copy` functions to this hierarchy of classes. It's a good thing we did it now before the interfaces became stable and adding functions became a royal pain, causing recompilation of calling code. Or did we?

The problem with all this reasoning is that it has little to do with whatever you're ultimately trying to achieve in your program. So many people who think more about what their program does than about the way their programming language works won't see this coming, and do something wrong. No, it *doesn't* mean that the people primarily focused on the programming language are more productive than others. Of course they aren't uniformly less productive, either. Some of the people with a programming language focus spend most of their time choosing between `func(obj)` and `obj.func()`, some don't; it depends on the person, and people can change, further complicating matters.

Of course this problem exists with all computer languages and, more generally, with all software systems, and, more generally, with all formal systems. Basically there's the ice - the things compatible with human common sense - and the cold water under it, into which you can fall when the rules governing the system contradict your common sense. So what's so special about C++?

Well, in C++, the very thin ice is covered with paint, and the water is deep enough to drown.

Inheritance -- what your mother never told you

Note: section names are copied verbatim from the FAQ. The document is not based on the assumption that it was your mother who told you the other things about inheritance.

[23.1] Is it okay for a non-`virtual` function of the base class to call a `virtual` function?

FAQ: Sometimes it is. Suppose you have a class `Animal` with a non-`virtual` `getAwfullyExcited()` method. In your system, all animals do it similarly:

```
void Animal::getAwfullyExcited()
{
  makeExcitedNoises(); // all animals make different noises
  cout << "And here comes the dance!" << endl; // all animals always warn people before they dance
  danceExcitedly(); // all animals dance differently
}
```

So there you have it - an orderly hierarchy of animals, each getting excited according to the standard procedure in its own unique way.

FQA: The trouble with this whole thing in C++ is that you can't easily tell which functions can be overridden and which can't, and what "override" means.

The default is "can't" - you have to explicitly say `virtual` to enable overriding. Unless you derive your class from a base class having `virtual` functions - when you override those functions, the `virtual` keyword becomes optional. But even if the function in the base class is *not* virtual, you still *can* override it - except that now the binding is static: if you call a function through a pointer statically typed as `Base*`, the base class implementation is called, but if you call it through a `Derived*`, the derived class implementation is called. So it's a different kind of "overriding". And naturally you can have more than one link in the chain of derived classes.

Back to our question, it is generally OK to have a common base class implementation call functions defined in the derived classes - in many cases that does exactly what you want. However, in the specific case of C++ `virtual` functions it may become hard to figure out which functions are `virtual` and which are not.

So it's sometimes better to separate `Animal` (having only pure virtual functions) and `EmotionalBehavior` (having methods taking an animal and calling its functions to implement various common rituals). No, it's probably *not* a good idea to add this rule to your local Coding Conventions Document - if the class is relatively small and simple, there's no point in creating more code by splitting it to several classes. The thing is that C++ classes with *many* methods, some virtual and some not, especially when they are part of complex hierarchies with lots of overriding of both kinds, end up making people wonder why this piece of code was called. Why, why, why?..

[23.2] That last FAQ confuses me. Is it a different strategy from the other ways to use `virtual` functions? What's going on?

FAQ: Yes, there are two different strategies related to the use of `virtual` functions. In the first case, you have a common non-`virtual` method in the base class, but use `virtual` methods to implement the parts which do differ. In the second case, you have `virtual` methods implemented differently in derived classes, but they call common non-`virtual` methods in the base class. These common bits can also be implemented somewhere else - not necessarily in the base class.

Sometimes you use both strategies in the same class - one for some of the methods, the other for other methods. That's OK, too.

FQA: Well, sort of, yeah. Is it just me or did we really learn nothing at all? This lengthy discussion follows quite directly from first principles. `virtual` functions allow different derived classes to implement a method differently. All functions, including non-`virtual`, can be called once or more from various places. Does something totally obvious from these rules deserve the pompous name "strategy"?

No, it's not just the FAQ. This is very common in the "software engineering" community, especially in the C++ subculture. Let's look at an example. You've probably seen methods that create new objects of classes derived from a common base. Such methods are useful because their caller doesn't have to be aware of the different derived classes. For example, you can have a `createMovieReader(filename)` method that checks the file type and creates an `MPEGReader` or an `AVIReader` or whatever, and whoever calls `createMovieReader` doesn't have to care about the many different kinds of readers. Is this worth a special term accompanied by a whole discussion? Well, it has a term - it's the Factory Method Design Pattern. Sometimes you have an abstract base class with nothing but Factory Methods. This has a name of its own, too - it's the Abstract Factory Design Pattern.

The names used by people reveal a lot about them. For example, the people living close to North Pole have a two-digit number of names for "snow". Clearly, the reason is that their visual diet is composed primarily of snow, so they know a lot about the different kinds of snow and never confuse them. But trees - those they come across once in a lifetime. No point in having many names for trees. A tree, you know, that dark, high thing with messy stuff sticking out.

This tells us something about the intellectual diet of people calling trivial combinations of basic language constructs "strategies" and "patterns". Of course these people love C++ - look at all those different string classes, and all those ways to implement still new ones! So much snow to play with.

Disclaimer: my knowledge of anthropology is approximately zero. Therefore, it's better to consider the North Pole example above to be hypothetical than to make critical decisions assuming it's literally true.

[23.3] Should I use protected virtuals instead of public virtuals?

FAQ: Well, first of all avoid strict rules saying "never do this, always do that". As a rule of thumb, experience tells that most of the time virtual functions are best made public, with two notable exceptions. First, there are functions which are supposed to be called only from the base class, the way described in the previous two FAQs. And second, there's the *Public Overloaded Non-Virtuals Call Protected Non-Overloaded Virtuals* Idiom:

```
class Base
{
public:
  void overloadingTotallyRules(int x)   { butWeDontWantToOverloadVirtualFunctions_int(x); }
  void overloadingTotallyRules(short x) { butWeDontWantToOverloadVirtualFunctions_short(x); }
protected:
  virtual void butWeDontWantToOverloadVirtualFunctions_int(int);
  virtual void butWeDontWantToOverloadVirtualFunctions_short(short);
};
```

This solves an important problem: overloading totally rules, but we don't want to overload virtual functions. The *Public Overloaded Non-Virtuals Call Protected Non-Overloaded Virtuals* Idiom makes life easy for class users *and* the implementors of derived classes! Clearly you, the author of the base class, absolutely *must* clobber your code with this nonsense to make everyone's life better! Think about the reduced maintenance costs and the benefit of the many, you selfish code grinder.

FQA: Here's a simple answer to your question, trivially following from the definitions and without the need to create a taxonomy of rules and exceptions. C++ access control is for clarity: you want to make it clear which parts of the class are supposed to be used and which are just implementation details. This can make it easier to understand your code and prevents people from using the bits that you might want to change later.

Therefore, if you don't think that anyone outside of the class hierarchy is going to use a `virtual` method, make it `protected`. Otherwise, make it `public`. In particular, you can start with `protected` and change it to `public` if it turns out you were wrong. That's all. Why is any special case of this particularly interesting?

As to the *One Kind Of Function Calls Another Kind Of Function Idiom* - remember the snow example? Well, now we are deep inside a pile of snow, coining names for each individual snowflake. I wonder if there's a *Function Is Passed A Parameter To Configure Its Operation Idiom*. No, there probably isn't - after all, it's not Object-Oriented.

More importantly, the "idiom" is a non-solution for the problems with overloading. All we got is extra layers of code piling up. Ultimately, the people will have to read this code to figure out what the program does, and the layers of indirection adding no functionality at all will confuse them and occupy their short-term memory instead of other, actually useful details. Why not simply *avoid* overloading in this case?

Talking about "reducing maintenance costs" doesn't by itself actually reduce any maintenance costs, you know.

[23.4] When should someone use private virtuals?

FAQ: Probably never. It confuses people, because they don't think `private` virtuals can't be overridden, but they can.

FQA: The FAQ is right - the behavior of `private virtual` is ridiculous; it compiles, but for the wrong reasons. Note that there are many, many more things in C++ that primarily confuse people. One excellent reason to avoid C++ altogether. I'm not joking. Nothing funny about it. Well, maybe it is funny that C++ developers are confused for a living, but that's a cruel kind of humor.

[23.5] When my base class's constructor calls a `virtual` function on its `this` object, why doesn't my derived class's override of that `virtual` function get invoked?

FAQ: Suppose you have a base class called `Base`, calling a virtual function `f` in its constructor. Then, when objects of a derived class called `Derived` are created, `Base::virt` is called from `Base::Base`, not `Derived::f`.

The reason is that when `Base::Base` executes, the object is still of type `Base`. It only becomes an object of type `Derived` when the code in `Derived::Derived` is entered. If you wonder why C++ works this way, consider the fact that `Derived::f` could access uninitialized members of the class `Derived` if it could be called from `Base::Base`, which runs before `Derived::Derived` initializes the members of `Derived`.

Luckily, C++ doesn't let this happen, preventing subtle errors!

FQA: Here's what actually happens. `Derived::Derived` calls `Base::Base`, and then it sets the vptr to point to the `Derived` vtable. Setting the vptr before calling `Base::Base` wouldn't work, because `Base::Base` sets the vptr to point to the `Base` vtable. `Base::Base` doesn't know that it's called in order to ultimately initialize a `Derived` object; the code of `Base::Base` and `Derived::Derived` is compiled separately *(correction)*. That's what "the object is still of type `Base`" really means.

Now, there's a valid argument against describing the mechanisms of the implementation in order to explain the behavior of a system. The system is built by people for other people. Its behavior is supposed to make sense. If it doesn't, and can only be explained by looking into the implementation instead of the needs of the user, it's a problem in the system. Sometimes such problems seem to be inevitable. But whenever possible, it's best to discuss why a certain behavior is reasonable in the situation as perceived by the user, and only talk about the implementation when absolutely necessary. However, this argument is rarely applicable to C++.

First, C++ makes very little sense from the user's perspective. It only makes sense from the perspective of the language designer, provided that several axioms of questionable value are added to it (such as "a language must look compatible with C, although it doesn't have to really be compatible", "all built-in types should be those found in C, unless they are a new kind of pointer", etc.). For example, it is reasonable that `Base::Base` can't call `Derived::f` (well, sort of - it depends on what you think about the way C++ handles object construction in general). But is it reasonable that `Base::Base` *can* call a virtual method `Base::f` by simply saying `f()`? How often is that what you want? This question is irrelevant, because in C++, if something becomes technically possible as a side-effect of some language mechanism, it tends to be legal. For example, why does `private virtual` from the previous FAQ compile? The answer is simple: why not?

The second reason to describe behavior in terms of implementation is that run time errors cause C++ programs to crash in ways undefined at the semantical specification level. It is theoretically possible to read the entire code of the program and find the semantically illegal code. In practice, you'll have to find the error by looking at the execution of the program (when you are lucky) or at a snapshot of its final state before the death (when you're not), and trying to understand how things can behave this way. And you can't do that without understanding the implementation.

[23.6] Okay, but is there a way to *simulate* that behavior as *if* dynamic binding worked on the `this` object within my base class's constructor?

FAQ: Yes. It's an Idiom, of course. Namely, the *Dynamic Binding During Initialization Idiom*.

One option is to have a `virtual init` function to be called after construction. Another option is to have a second hierarchy of classes, which doesn't always work, but... (sorry, I couldn't make it through all the code listings. If you like lots of hierarchies of classes and find this solution interesting, please follow the link to the FAQ's answer).

The first approach has the problem of requiring an extra function call upon initialization. We can rely on the self-discipline of the programmers (the self-discipline is especially important when exceptions can be thrown by `init` - make sure you release the allocated object properly), or we can wrap the construction and the `init` call in a single `create` function returning a pointer to the object. The latter rules out allocation on the stack.

FQA: We seem to be making progress. Let's see: we got rid of the allocation on the stack, so we no longer need to know the `private` members of classes. And if we had garbage collection, exception safety would no longer be a problem, either. Too bad C++ classes and memory management can't be changed. Or can they?

How about trying another programming language?

[23.7] I'm getting the same mess with destructors: calling a `virtual` on my `this` object from my base class's destructor ends up ignoring the override in the derived class; what's going on?

FAQ: Again, you're being protected by the compiler. Protected from yourself! You could access members that were already destroyed if the compiler let you do what you want.

When `Base::~Base` is called, the type of the object is changed from `Derived` to `Base`.

FQA: Thanks for the protection. At least the behavior is symmetrical.

Protect me from access to destroyed objects through dangling references in *the general case* next time, will you?

[23.8] Should a derived class redefine ("override") a member function that is non-`virtual` in a base class?

FAQ: You can do that, but you shouldn't.

Experienced programmers sometimes do that for various reasons. Remember that the user-visible effects of both versions of the functions must be identical.

FQA: Here's a trade secret: not so experienced programmers do that, too. For the umpteen time: *why does this compile*? This particular case doesn't seem to be an unintended side effect; it's a feature with elaborate design (there's all this nonsense with `using` names from the base class shadowed by functions with the same name, etc.).

The redefinition-looking-like-an-override is overloading on steroids: it's even less useful and has even higher obfuscation potential. And it's only one of the zillions of various *name binding rules* - the bits of C++ making it impossible to decipher what code is actually called by `f(x)`.

[23.9] What's the meaning of, `Warning: Derived::f(char) hides Base::f(double)`?

FAQ: What do you think it means? You're going to die.

It's like this: `Derived::f(char)` doesn't override `Base::f(double)` - it hides it. In other words, it makes it impossible to call `Base::f(double)` via a `Derived*`. At the same time, `Derived::f(char)` can not be called via `Base*`.

FQA: "You're going to die", warns the FAQ. Well, according to a popular theory, all of us are. However, this interesting fact doesn't seem to belong here. C++ surely is depressing, but I'm not aware of any data showing a causal relationship between using C++ and suicide or lethal brain damage.

No, really, why is hiding a name *a lethal bug?* Sure, it makes the program more obscure, but, duh, this is C++. The worst thing that can happen is that someone will call `Base::f` with a `char` and `Base::f(double)`, not `Derived::f(char)` will get called. So what? Similar things happen with overloading without any inheritance involved. Do you really know the implicit conversion rules between different types, and can predict what happens when you pass an `int` to an overloaded `f` function which only has a `char` and a `double` version? I bet you can't do it when things get a little bit more complicated. No normal person can.

C++ overloading is a nightmare. Using it is naive or stupid, depending on one's experience. Many languages, both statically and dynamically typed, don't have compile time overloading based on argument types. Have you ever heard a programmer using such a language complain about it, or become very enthusiastic about C++ overloading? What problem does overloading solve? OK, suppose it improves clarity (a *very* questionable claim). Do you really think that overloading should come together with a ton of implicit conversion rules, and a type system with a ton of special cases (cv-qualifiers, pointers/arrays/references, single & multiple inheritance, templates...)? When overloading does come with the extra ton of rules, does it *still* make the program more clear? What's that? "Compile time polymorphism", you say? You mean C++ templates? I see what you're up to. Happy debugging, pal.

[23.10] What does it mean that the "virtual table" is an unresolved external?

FAQ: Well, as you know, "unresolved external" means that there's a function or a global variable that your code declares and uses, but never defines. A virtual table is a global variable declared implicitly by a typical C++ implementation for each class with at least one virtual function.

Now, normally when you forget to define a virtual function, you'll get an "unresolved external" error saying that this function is missing. But in many implementations, if you forget the *first* virtual function, the whole virtual table will become "unresolved". That's because these implementations define the virtual table at the translation unit where the first virtual function is implemented.

FQA: GAAA!! So THAT'S why you have to define `static` class variables explicitly in your .cpp file, but *don't* have to define vtables explicitly! This is sooo STUPID! I'm shocked. Wait a second. Let me recover.

OK, here's how it works. In C++, there's really no such thing as "a header file defining the class" and "a .cpp file implementing its functions". It's just a convention. According to the rules, the definition of the class members can be spread across several "translation units", and each "translation unit" can in turn be spread across several files `#including` each other. This is inherited from C, and interacts badly with the new features in C++.

For example, consider those virtual functions. You probably need a table of those, which is similar to a global C variable. Of course you don't want to have the C++ programmer implement the table manually, the way they'd do it in C - or else what's the point of the `virtual` keyword? But *where* should the vtable be implemented? In the .cpp file of a class? *Where's that?*

With vtables, there's a "solution". After all, we don't need a vtable unless we have some virtual functions, do we? Well, then, one of these functions has to be the first one. Why not place the vtable near the definition of that function? This is a good place, because it won't get `#included` twice (or the user will get a "multiple definition" error anyway).

And then there are the `static` class variables. Where do we stuff those? Well, um, there seems to be no good place at all; a class could consist of a single `static` member. No anchor in the sea of source code floating in the file system. Oh, well, we'll have the user choose a place for a duplicate `Type Class::member_name;` variable definition. A little bit more typing is not that bad.

I wonder what they do with `virtual` functions implemented in the body of a class though. I always wondered about those. Of course they can generate many vtables and throw away all copies but one the way they do with templates. But that won't work with a linker only supporting C. Maybe they didn't allow to implement non-`inline` functions in the body of a class before the brave decision to add features to the linker. Anyone knowledgeable about the history of the subject is welcome to enlighten me. However, the knowledge of the history of the subject can't possibly make this whole business seem less stupid to a language user.

[23.11] How can I set up my class so it won't be inherited from?

FAQ: You can have the constructors `private`. The objects will have to be created by `public` functions delegating to the constructor.

Alternatively, you can comment the fact that you don't want the class to be inherited from: `// if you inherit from this class, I'll hunt you down and kill you.`

There's a third solution (**WARNING** - this one can rot your brain): you can inherit your class from a `private virtual` base class, which has a `private` constructor and declares your class a `friend`. This way, if someone tries to inherit from you, they'll need to directly call the base class of the constructor, which won't compile, since the constructor is `private`. This can add an extra word of memory to the size of your objects though.

FQA: C++ doesn't have `final`. It probably isn't a big deal, especially considering the fact that if it *did* have `final`, it would be little more than a comment, just like `private` is little more than a comment. So the comment-instead-of-a-keyword is probably the best approach.

The other approaches yield cryptic compile time error messages. Many people in the C++ community believe that a cryptic compile time error message is a good thing. This is probably reasonable if you only care about theory ("early fault detection is good"), and ignore practice (*easy* fault detection is good, and running a program *should* be easier than deciphering strongly encrypted C++ compiler error messages, which tend to come in large cascades).

Having no real experience with other languages helps one to stick to this bizarre point of view. C++ compiles very slowly, so people who never worked with anything else *don't* think that running a program is easy. C++ code is also hard to debug, so people develop a fear of run time errors.

[23.12] How can I set up my member function so it won't be overridden in a derived class?

FAQ: Use a `/* final */` comment instead of a `final` keyword. It's not a technical solution - so what? The important thing is that it works.

FQA: So why does C++ have `private`? It's nothing but a comment recognized by the compiler.

The real answer is as follows. If a function is not `virtual`, it shouldn't be overridden, as the FAQ itself has already explained. So the lack of a `virtual` keyword is effectively equivalent to a `final` keyword. Now if C++ didn't allow to override non-`virtual` methods, and if it would consistently require to use the `virtual` keyword in all virtual function declarations, it would be pretty clear which functions shouldn't be overridden in most cases.

If you look for non-technical solutions to overriding problems, consider banning non-`virtual` overrides in your organization. I didn't give this one a deep thought; quite frequently when you try to "fix" C++ by banning some of its features, but not others, your rules backfire. And I can't know whether coding conventions work in your organization at all (believe me, there are *huge* and *very disciplined* organizations where coding conventions are *not* really followed, and their primary effect is programmers feeling anger or guilt; another nice option is people mindlessly following "best practices", wreaking havoc in an orderly way). All I'm saying is that non-`virtual` override seems both useless and largely independent of the rest of the language at first glance, so not using it looks like a good thing.

Inheritance -- multiple and virtual inheritance

This section is about multiple inheritance. While inheritance and `virtual` functions are among the most useful (that is, the least useless) C++ features, C++ multiple inheritance is at the other end of the spectrum.

[25.1] How is this section organized?

FAQ: The FAQ section contains both "high-level" issues (the meaning & purpose of the language constructs) and "low-level" issues (the detailed semantics & implementation of the language constructs), in that order. Be sure to understand the high-level stuff before delving into the low-level stuff.

FQA: This FQA section is organized exactly like the other FQA sections - by copying the structure of the FAQ sections.

The FAQ's note about the two levels of discussion is equally applicable to all C++ features/programming languages/software systems/formal definitions in the world. But it's located here and not elsewhere for a reason. The reason is that C++ multiple inheritance makes very little sense even by C++ standards. In other words, the cases where you can express a useful high-level idea using the actual features used to implement multiple inheritance in C++ are rare. So people end up not using it, or abusing it, but fail to use it "right", because it's unclear what "right" means in this context.

The FAQ apparently believes that this situation can be fixed by explaining what "right" means. Let's sit back and watch.

[25.2] I've been told that I should never use multiple inheritance. Is that right?

FAQ: THESE PEOPLE REALLY BOTHER ME!! How can they know what you should do without knowing what you want done?!?!

FQA: This rage may provoke some sympathy. We've all met people who don't seem to be doing anything useful themselves and compensate for it by getting in others' way and telling them how to do their job, their ultimate goal apparently being to stop any useful work around them. In particular, language features which should really *never* be used are rare. However, language features which should be used really rarely and with careful consideration are more common, and C++ multiple inheritance is one of them.

There are two kinds of problems with multiple inheritance - "static" and "dynamic".

The "static" problems have to do with compile-time name lookup. Suppose you derive a class C from classes A and B, and both have a method called `name`. Which one will get called when someone calls `name` using a pointer to a C object? If you override `name` in the class C, did you override `A::name`, `B::name` or both? What happens if the two `name` functions accept arguments of the same type? What happens if they don't?

The "dynamic" problems have to do with the way objects of the class C are actually built, and the run time effects related to it. Basically, a C object will contain an A sub-object, and a B sub-object, and those two will be completely unrelated. So if you have a pointer to a C object, and you (silently) upcast it to A*, and write code assuming that you get an object which is derived from both A and B, and cast the A* to B*, the program will crash or worse. What you should have done is first cast the object to C* and *then* to B*, since without knowing the definition of C, there's no way to figure out the location of the B sub-object given the location of the A sub-object. This is one trap even a pretty experienced C++ programmer can fall into.

You can memorize the sharp edges, or you can stay away from the whole thing. Pick your poison.

[25.3] So there are times when multiple inheritance isn't bad?!??

FAQ: Sure! Sometimes (but not always) using multiple inheritance will lower all kinds of costs. If this is so in your case, use it! If it isn't, don't blame multiple inheritance - "good workmen never blame their tools".

FQA: No, there are no such times, it's a poorly designed language feature. But it could be that sometimes the other options available in C++ are even worse.

While we're at it, let's clarify the whole blame issue. If you *choose* to use a tool not suitable for your job, you shouldn't blame the tool. Well, actually, you should blame the vendor of the tool if it was advertised as something it wasn't. Was C++ ever advertised that way, for example, what about support for object-oriented programming?

Well, we don't even need to discuss that, because a programming language is not exactly a tool. It is more accurately described, well, as a *language*. The key difference between tools and languages in the context of "blame" is *choice*. You probably don't choose to speak English - you do so in order to communicate with all the other people speaking English. When a bunch of people do something because other people do it, too, it's called "network effects". For example, if you want to work on a project for reasons having nothing to do with computer linguistics, and the project uses C++, you'll have to use C++, too. No choice.

So that's the difference between a language and a tool. Still, you wouldn't blame English because it's so hard to learn or inconsistent or whatever, would you? Well, the difference between a programming language and a natural language is that the latter is, um, natural, so there's nobody to blame, while the former was actually designed by someone (well, usually). The other difference is *the cost of an error*. People usually recover from bad English, computers tend to be less tolerant.

And this is why you seem to have more ground to "blame the language" than to "blame the tools" in the general case. Of course you may not like the whole attitude of "blaming" things, etc.; everybody is free to feel any way they feel like or something. But that has nothing to do with being a "good workman" (which itself has an irksome sound to it, mind you).

[25.4] What are some disciplines for using multiple inheritance?

FAQ: There's a long answer saying 3 things. First, you should normally do it to achieve polymorphism, not base class code reuse. Second, the classes you multiply inherit from should normally be pure abstract. Third, you should consider using the "bridge pattern" or "nested generalization" - alternatives to MI described below.

FQA: The guidelines are pretty good, except for maybe "nested generalization", which is really a way to work around the deficiencies of the C++ object system rather than a reasonable way to model anything.

If you like "disciplines" without any reasoning having to do with the actual problem at hand, here's some more for you. The FAQ's guidelines are a special case of "don't use designs which would only work in one programming language" (footnote: *especially* if that language is C++). Specifically, the FAQ's guidelines pretty much summarize the *rules* for using multiple inheritance in Java, so your design would be implementable in at least two languages, which is a good sign. The reasoning behind the avoid-designs-tied-to-one-language rule is that if something is really good, many languages would have it, and if your design depends on something only available in one language, it's probably bad because it probably depends on a bad thing. This is the point where people loving the unique feature of language X scream that this reasoning is completely *moronic*, but we already knew that, because we promised our reasoning wouldn't refer to the specific problem at hand, which isn't very bright by itself.

If you're into real reasoning and not just "disciplines", one nice thing about having the base classes pure is that this way, you don't have to think about a whole class of questions related to reimplementation of methods. For example, if you inherit a `RectangularWindow` from a `Rectangle` and a `Window`, and `Rectangle` isn't pure and it has a perfectly good `resize` method, is this method still good for `RectangularWindow` or do you want it to resize the window, which the implementation in the base class obviously won't do? And what if you can't really override the `Rectangle::resize` method because it isn't `virtual`? The problem with reusing code from the base class is that multiple inheritance frequently breaks that code.

However, following these guidelines won't necessarily eliminate the problems with multiple inheritance mentioned above.

[25.5] Can you provide an example that demonstrates the above guidelines?

FAQ: There's a very long discussion of an example with different kinds of vehicles having different kinds of engines. The FAQ proposes to use MI, or the "bridge pattern" or "nested generalization". "Bridge pattern" means that vehicle objects keep pointers to engine objects, and users can pass many different kinds of engine to an object of the same vehicle class. "Nested generalization" means that you have many classes derived from `Vehicle` (like `Plane`), and then for each such derived class there's a bunch of classes derived from it to represent the different kinds of engine (like `OilPoweredPlane`). Trade-offs are discussed in great detail.

FQA: The "bridge pattern" (a fancy name for the special case of aggregation when your member object has `virtual` functions) looks good here, since, um, a vehicle has an engine and stuff. And hence multiple inheritance looks wrong, since an `OilPoweredPlane` isn't a kind of an `OilPoweredEngine`.

I don't feel like arguing with the FAQ's lengthy statements, since the issue isn't worth it. The cases when you deal with the definition of non-trivial object models are relatively rare. And when you do it, you have enough time to consider what stuff you really want the model to support, and then think about the different possibilities to define the model and check if each possibility really supports that stuff. I think that trying to memorize special cases (call them "patterns" or whatever) of object models is basically like trying to formalize common sense, which doesn't really work.

Are you still with me after this blasphemy? Then let's look at one non-problem mentioned by the FAQ - the fact that with aggregation ("bridge pattern"), you can't specialize algorithms such that a specific combination of vehicle and engine exhibits a special behavior. In languages which support multimethods, doing that is trivial (multimethods are like `virtual` functions, but they are dispatched at run time based on the types of all arguments, not just the first argument). And in C++, you can emulate multimethods using double dispatching (ugly, especially when it becomes triple, quadruple and other such kinds of dispatching, but still possible).

[25.6] Is there a simple way to visualize all these tradeoffs?

FAQ: Here's a matrix with cute smilies for ya. Just don't apply it naively.

Cute matrix omitted to avoid copyright problems, as well as cuteness problems

FQA: **WARNING:** there's no known way to represent common sense in a tabular form at the time of writing. Therefore, if you choose to store the cute matrix anywhere in your brain, you do it at your own risk.

[25.7] Can you give another example to illustrate the above disciplines?

FAQ: Yes - consider the case with land & water vehicles, when you also need to support amphibious vehicles. This case is more "symmetric" than the previous example, so multiple inheritance becomes more preferable. Still, you have to make sure you really want it by asking various questions (for the list of questions, follow the link to the FAQ).

FQA: Um, "symmetry" is an interesting aspect of this to focus on, but anyway, an amphibious vehicle is both a land vehicle and a water vehicle, while an oiled powered plane is a plane, but is not an oil powered engine. So yes, multiple inheritance seems more appropriate, and yes, it's wise to think about the things you ultimately want your object model to support before defining it.

We'll use the opportunity to show how to model this problem effectively using multiple inheritance (implementation of multiple interfaces by the same class) without really using C++ multiple inheritance (and thus avoiding some of its problems). I'm not saying that this always better than real C++ multiple inheritance, just that it sometimes can be.

```
class AmphibiousVehicle {
  class WaterVehicleImpl : public WaterVehicle {
    WaterVehicleImpl(AmphibiousVehicle* p) { /* save p */ }
    ...
  };
  // similarly, there's a LandVehicleImpl class derived from WaterVehicle
  WaterVehicleImpl _water;
  LandVehicleImpl _land;
public:
  AmphibiousVehicle() : _water(this), _land(this) {}
  WaterVehicle& getWaterVehicleIF() { return _water; }
  LandVehicle& getLandVehicleIF() { return _land; }
};
```

This way, you write more code than with multiple inheritance, which is bad. It gets even uglier if you want to simulate virtual inheritance, which is bad if `WaterVehicle` and `LandVehicle` inherit from a non-abstract base class `Vehicle` (not necessarily recommended by itself). And you have to call `get` functions instead of implicit upcasts, which may be considered good or bad. And there are no problems such as collisions between names of the members of the base classes (which is good).

[25.8] What is the "dreaded diamond"?

FAQ: It's when there are circles in the inheritance graph. Here's the simplest case: `Derived1` and `Derived2` are inherited from `Base`, and `Join` is inherited from both `Derived1` and `Derived2`. The circle in the graph may look like a diamond if your imagination works that way.

The problem is that `Join` objects have *two* `Base` sub-objects, so each data member is kept twice. Which is why the diamond is called "dreaded".

The resulting ambiguities can be resolved. For example, when you have a `Join` object and refer to its `_x` variable inherited from `Base`, you can tell the compiler which one you mean using `Derived1::_x` or `Derived2_::x`. When you upcast from `Join*` to `Base*`, you can pick one of the two `Base` sub-objects by first casting the pointer to `Derived1*` or `Derived2*`. But this is almost always not the right thing to do. The right thing to do is usually to tell the compiler to keep a single sub-object.

FQA: Most C++ programmers out there don't understand why would anyone say `(Derived1*)pJoin` in a context where a `Base*` is expected. This by itself is a good reason to avoid having two sub-objects of `Base` in `Join`.

If you feel that things are getting pointlessly complicated at this point, it may be an indication of good taste.

[25.9] Where in a hierarchy should I use virtual inheritance?

FAQ: At the top of the dreaded diamond - when you derive from `Base`, you should say:

```
class Derived1 : public virtual Base { ... };
class Derived2 : public virtual Base { ... };
```

Note: when you define `Join`, you can't convince the compiler to keep a single `Base` sub-object - it will do whatever the definitions of `Derived1` and `Derived2` tell it to do. That is, when you define the classes derived from `Base`, you must plan ahead to support circles in the inheritance graph.

FQA: Let's put aside the question whether the support for both options - one and two `Base` sub-objects - is a good thing, and concentrate on the way C++ gives you to choose between these options. Doing it "at the top of the diamond" is annoying, because you have to think about the entire hierarchy when you define the classes close to its top. That is, either *all* derived classes will have several `Base` sub-objects or *all* of them will have one (forcing the users and the implementers of derived classes to deal with the problems of virtual inheritance).

In general, it is a special case of the generic C++ principles of specifying everything in terms of types and their attributes, as well as having the user deal with the low-level details related to underlying language feature implementation.

[25.10] What does it mean to "delegate to a sister class" via virtual inheritance?

FAQ: If you have a diamond-like hierarchy with virtual inheritance, and `Base` has two virtual functions f and g, then `Derived1` can implement f, `Derived2` can implement g and `Derived1::f` can call `Derived2::g` by simply saying `g();` or (more verbosely and equivalently) `Base::g();` - that is, without knowing anything about the existence of `Derived2`.

This is a "powerful technique".

FQA: "Powerful". What exactly can you do this way that can't be done equally well or better in ways more clear to the average developer?

What's that? You say that you only care about the enlightened wizards (variant: the set of wizards consists of a single person - yourself), not the mediocre droids from the rank-and-file? Well, I'll leave the interesting discussion of your personality aside. I'll leave it aside in order to point out that some of the people whose programming abilities I admire can't be bothered to learn the quirks of C++ anywhere near my level. My level, in turn, isn't itself anywhere near "complete" knowledge of this wonderful language.

[25.11] What special considerations do I need to know about when I use virtual inheritance?

FAQ: Usually virtual inheritance is a good idea only if the virtual base class and classes derived from it have little or no data.

BTW, even if they have no data at all, using virtual inheritance can still be better than non-virtual inheritance. For example, if you have two `Base` sub-objects (with no members), you can end up with two pointers to the different sub-objects, and comparing them would tell you that these are two different objects, which they aren't, at some level. Quote: "Just be careful - very careful".

FQA: Yeah. Be vewy, vewy caweful...

The FAQ's advice is a special case of its other advice about inheritance - data in base classes interacts badly with MI. And as the FAQ correctly points out, not having data in base classes doesn't solve all of the problems.

[25.12] What special considerations do I need to know about when I inherit

from a class that uses virtual inheritance?

FAQ: Derived classes call the constructors of their virtual base classes directly. In particular, when a virtual base class has no default constructor, you have to call its constructor explicitly in the initialization lists of the constructors of the derived class.

If the base class follows the FAQ's advice about not having data in virtual base classes, then the base class probably has a trivial default constructor and you don't have to care about the issue when you define derived classes.

FQA: If the base class follows the FQA's advice to avoid non-trivial constructors and use initialization functions when needed, you don't have to worry about initialization of derived classes with virtual base classes, either.

[25.13] What special considerations do I need to know about when I use a class that uses virtual inheritance?

FAQ: Don't use downcasts using the C-like syntax `(Derived*)pBase`. Use `dynamic_cast<Derived*>(pBase)`.

The answer is unfinished according to a "TODO" remark in it.

FQA: The problem seems to be that with virtual inheritance, the offset that must be added to `pBase` to make it `pDerived` depends on the classes *derived from Derived* (like the `Join` class from the "dreaded diamond" example). So the compiler can't generate code adding a constant offset, which is what C-style casts do when it comes to class hierarchies (upcasting and downcasting).

Why does the code silently compile to a wrong program, despite the fact that C++ already forced us to inform the compiler that we have virtual inheritance at the definitions of the classes involved in the cast operation (not the definition of the `Join` class which isn't necessarily visible at the context where the cast operation is compiled)? Why doesn't the compiler produce an error message or generates correct code as if we used `dynamic_cast`?

The answer is simple: in C++, the compiler compiles random meaningless things because it can't be bothered not to.

[25.14] One more time: what is the exact order of constructors in a multiple and/or virtual inheritance situation?

FAQ: So and so.

FQA: I don't want to summarize it, because why would anyone want to know that? Well, except maybe to suppress stupid compiler warnings about the orders of things in initialization lists not matching the actual order of construction.

Well, why would you use initialization lists?

[25.15] What is the exact order of destructors in a multiple and/or virtual inheritance situation?

FAQ: The reverse order of construction.

FQA: Right. But you probably shouldn't write code that depends on these things. Your colleagues may get annoyed.

How to mix C and C++

These questions are about mixing C and C++, which may be harder than you'd expect from the names of those languages, but easier than, say, mixing C++ and C++. Stay tuned.

[32.1] What do I need to know when mixing C and C++ code?

FAQ: You should check your vendor's documentation. Most frequently the rules are:

- You must compile `main` with your C++ compiler.
- You must link everything with your C++ linker.
- Your C and C++ compiler should be compatible, which probably means they should have the same vendor and version.

And you'll need to read the rest of this section so that your C functions can call your C++ functions and vice versa.

Or you can compile the C code with your C++ compiler - you may need to change the code, but you may also find bugs this way, so it's a good thing to do. Unless you don't have the C code in source form, of course.

FQA: You have little chances to successfully apply the rules unless you understand the underlying technical problem the rules try to address. The problem is in all the things in C++ that can not be translated to C straight-forwardly (mostly exceptions), and the things which can be translated in *several* ways (initialization of global variables before `main` and finalization after `main`, mangling names of overload & template functions, virtual function calls, constructor prototypes, layout of derived classes, RTTI - this list is quite large). Many languages which are easy to mix with C have such features. However, unlike C++ they also come with a formal or a de-facto standard defining the ABI (application binary interface) or a source-level interface for C interoperability. The C++ standard doesn't bother.

Most of these things can only cause problems when a particular C++ function is explicitly called. This kind of things is addressed in the rest of the questions in this section. However, the global initialization & finalization sequences are never explicitly called - hence the requirement about compiling `main` and linking the program with the C++ compiler. As to the need to use C and C++ compilers from the same vendor - this is true in theory, but in practice C compilers for a given hardware/OS configuration will interoperate smoothly. So will the C++ compilers as long as the C subset of the calling conventions is involved. However, this requirement is almost always a must when *mixing C++ and C++* - for example, when third-party libraries with C++ interfaces are involved. Think about it: mixing C and C++ is easier than mixing C++ and C++. Isn't this *amazing*?

This situation is one excellent reason *not* to follow the FAQ's advice to compile your C code with a C++ compiler: C code is more portable. There are other reasons to keep C code in C, such as compilation time, better accessibility (there's no name mangling so functions bundled into a shared library are easier to call), etc.

[32.2] How can I include a standard C header file in my C++ code?

FAQ: Like this: `#include <cstdio>`, and then `std::printf("I like std::!\n")`. If you don't like `std::`, get over it. That's the way standard names are accessed.

If you compile old C code with a C++ compiler, the following will also work: `#include <stdio.h>`, and then `printf("No std::!\n");` - all due to the magic of namespaces.

If you want to include a *non-standard* C header, see the next questions.

FQA: Um, if `printf` and `std::printf` both work, what's there to get over? `printf` is so standard that there seems to be little point in mentioning it over and over again. As to the "magic of namespaces", this particular case doesn't really seem to have anything to do with it. For some reason, the global unmangled `extern "C" printf` is also made accessible via `namespace std` by the C++ standard. What's so mysterious or amusing here? Perhaps the FAQ meant "the magic of standards".

If you want to include a non-standard C header, basically you'll have to tweak them the same way your compiler vendor tweaked the standard C headers.

[32.3] How can I include a non-system C header file in my C++ code?

FAQ: Like this:

```
extern "C" {
#include "foo.h"
}
```

If `foo.h` is your header, you can change it to make inclusion from C++ easier.

FQA: The reason you have to do this is that C function names are not *mangled* - in C, there's no overloading, so `printf` is known to the linkers, debuggers, etc. as `printf`. But in C++ there may be several functions with the same name. So the compiler has to make up a unique name using an encoding of the argument types. For example, the GNU C compiler generates an assembly function called `_Z6printfPKc` from C++ source code defining `int printf(const char*)`. Different C++ compilers will do the name mangling differently - one of the many reasons making them incompatible with each other.

Theoretically there may be more differences between C and C++ functions, and `extern "C"` is your way to tell your C++ compiler "these are C functions, deal with all the differences". In practice, the problem is name mangling. Too bad there's no `extern "C++ compiled with a different compiler"`.

[32.4] How can I modify my own C header files so it's easier to #include them in C++ code?

FAQ: Like this:

```
#ifdef __cplusplus
extern "C" {
#endif
void foo();
void bar();
#ifdef __cplusplus
}
#endif
```

Ew, macros are *evil*, wash your hands when you are done.

FQA: Together with the usual `#ifndef,#define,#endif` trinity, we've just used 7 preprocessor directives to define a single interface. And these seven directives contain zero information specific to that interface. And they don't help the compiler to do things compilers of other languages can do, like locating the implementation of the interface.

If you want to wash your hands after each preprocessor directive you touch in C++ and have some time left to do anything else with those hands, you'll have to work in a bathroom.

[32.5] How can I call a non-system C function `f(int,char,float)` from my C++ code?

FAQ: Prefix its prototype with `extern "C"` when you declare it. You can declare a whole bunch of C functions by surrounding the declarations with an `extern "C" { ... }` block.

FQA: Yeah, we've been through this already. It doesn't matter whether a declaration is in a header file or not. Neither C nor C++ syntax is aware of header files or other preprocessor-related things. Header files are just an automated copy-paste mechanism.

[32.6] How can I create a C++ function `f(int,char,float)` that is callable by my C code?

FAQ: Prefix the declaration and the definition with `extern "C"`. You can't have more than one `f` C-callable function since C has no overloading.

FQA: Oh, how simple! *And what if the function throws an exception*? You didn't think you were going to escape *that* easily, did you?

I've just tried this with the GNU C and C++ compilers. When a C++ function calls a C function which calls a C++ function which throws an exception, you can't even catch it at the first C++ function, not to mention disposing the resources allocated by the C function.

So, make sure you catch all possible exceptions in your C-callable C++ functions. By the way, C++ exceptions can be of any built-in or user-defined type, and you can't catch an arbitrary exception and check what kind of exception it is at run time, and `operator new` can throw exceptions. Enjoy.

[32.7] Why is the linker giving errors for C/C++ functions being called from C++/C functions?

FAQ: You probably forgot `extern "C"`, so the linker looks for a mangled C++ name instead of an unmangled C name.

FQA: Quiz: does a typical C++ linker *try to check* whether the unmangled C name is defined, and if in fact it is, ask you something like "did you forget `extern "C"`?" Hint: if it actually did this simple thing, how frequently would this question be asked?

You see, one of the advantages of using C++ is that you get to work with mature, industrial-strength tool chains.

[32.8] How can I pass an object of a C++ `class` to/from a C function?

FAQ: You can use `class Fred` in C++ (`#ifdef __cplusplus`), and a `typedef struct Fred Fred;` otherwise. Then you can define `extern "C"` functions which accept `Fred*` (the FAQ contains two screens of code illustrating this point, including both ANSI and K&R C function prototypes).

Note that this way, C++ code will be able to tell whether two pointers to class objects point to the same object, and C code won't. That's because when a pointer to a base class object is compared to a pointer to a derived class object, the compiler may need to do some pointer arithmetics before the comparison. In C++, this is done implicitly when the expression `p == q` is compiled.

Note that if you convert pointers to objects of classes to `void*` and compare them, neither C nor C++ compilers will be able to do the right pointer adjustments.

FQA: *Please* don't follow this advice! This FAQ keeps telling how evil the preprocessor is, and then it proudly presents this *really nasty* scheme. Defining type names to mean different things based on a preprocessor flag is as close to "evil preprocessor abuse" as it gets. Especially with all these pointer equality subtleties involved (these are ridiculous by themselves - seriously, if you can shoot yourself in the foot by simply *comparing two pointers to objects*, how "object-oriented" is the language?).

Here's a pretty straight-forward solution: in the header file which is supposed to be used from C, declare a `struct FredObj` or something (just *use a different name* than `Fred`, so that people can at least figure out what each name means! Sheesh!). In the C++ implementation file, define the structure to hold a single member - a `Fred` object. This doesn't lead to any run-time overhead. The extra syntax needed for dereferencing is worth the benefits - you can compare pointers and have fun in safety.

And if you *really* need to return objects of classes derived from Fred - *just define a structure with a single member of type* `Fred*` *and never mind the tiny run-time overhead*. If you are using class hierarchies to implement functionality so simple that this tiny run-time overhead is comparable to the actual work done by the classes, *throw these class hierarchies away* and stop messing up the lives of your innocent users.

Why do these people have to make everything cryptic *and* dangerous?

[32.9] Can my C function directly access data in an object of a C++ `class`?

FAQ: Yes, if the class has no `virtual` functions or non-public members, and so do all objects it contains by value. The FAQ outlines the way inheritance and virtual functions are implemented at a level allowing you to do the pointer arithmetics in order to access member data from C in these clearly illegal cases.

FQA: *"Can"* may mean many things: "can do it with a particular version of C & C++ compilers", "can do it with all compilers which are actually out there", "can do it with any standard-conforming compilers", and even "should normally do it because provisions were made to make it easy".

The short answer is that you should only do it with the so-called POD types (which basically means "structures defined using C syntax" for people who are not professional language lawyers). The only reasonable cases when breaking the rules is not an entirely moronic act are (1) when you play around with the language to see what's inside, (2) when you have to retrieve data from classes with definitions only available in binary form (you may want to check if your actions are legal first) and (3) you are implementing a debugger or the like, in which case you're writing legitimately non-portable code.

In case (2), you could also ask "Can my C++ function directly access private data in an object of a C++ class". Most often it can if you add a `#define private public` preprocessor directive at the top of your `.cpp` file. This works quite portably and does not depend on the layouts of C++ classes in your particular compiler.

People who want their C code to directly access data of a C++ class object for "speed" or something probably don't have enough real problems. The artificial problems they create for themselves will teach them a good lesson pretty soon.

[32.10] Why do I feel like I'm "further from the machine" in C++ as opposed to

C?

FAQ: You are! C++ is a high-level language. In C, you can see where every clock cycle is spent; on the other hand, in C++ you can work at higher levels of abstraction and write more compact programs. Of course you can still write bad code - the idea is not to prevent bad programmers from doing it, but make it possible for the reasonable ones to write superior code!

FQA: What is this question doing here? Presumably people try to mix C and C++, and have an `extern "C"` function implemented in C++ throw an exception, or the initialization stuff before `main` never gets called, or they compare pointers to C++ class objects from C and it doesn't work, and they don't know why or how to even start figuring it out. The real question probably is "why do I feel like I'm underneath the machine, not just close to it as opposed to C"?

C++ is not a higher-level language than C. The damage caused by low-level errors is still not limited. You still have to think about pointers and object life cycles and integer endianness and many other things. But on top of that, there's a huge amount of things done implicitly, like global initialization and destruction, stack unwinding, base/derived classes pointer adjustment, and many more things - and all of them combine with the low-level errors into a single deep, wide tar pit with the programmer in the middle.

A good high-level language allows you to forget about many small details of program execution. A good low-level language allows you to control the many small details of program execution. C++ is not much of a high-level language, but it's not a very good low-level language either.

As to the remark about "seeing every cycle spent in C programs", I really believe that the FAQ author knows that you can't see that, since that's a pretty basic fact. You can't "see every cycle" spent in *assembly* programs in most cases - you have to know the exact target processor variant and the system configuration and a zillion other things. The FAQ is probably just being poetical.

But there's more to this remark than factual inaccuracy - it concentrates on a moderate problem, failing to mention an arguably more severe one. Consider the C++ code `p = obj.getVec().begin();`. The run-time of this code is unclear because it depends on whether `obj` is a value or a reference (the latter may be slower); in C it would be more clear. But there's another issue: is this code correct at all? If `getVec()` returns a reference to `std::vector` object, maybe it is correct, but if it returns it by value, it is certainly wrong. The compiler won't even warn you, and you won't notice the problem in the code without checking the definition of `getVec`. Not only is it hard to figure out how much time a C++ program runs, it is hard to even tell what it *does*, which is not supposed to be typical of high-level languages.

Pointers to member functions

This is basically about the lack of function objects and closures in C++.

[33.1] Is the type of "pointer-to-member-function" different from "pointer-to-function"?

FAQ: It is.

If you have a non-member function `void f(int)`, then `&f` is of type `void (*)(int)`.

If you have a non-`static` member function `void C::f(int)`, then `&C::f` is of type `void (C::*)(int)`.

FQA: Ahem. `::*)(` - line noise creeps in. There's more of it in the rest of this section.

Anyway, the reason the types *should* be different is that a member function accepts a hidden parameter - `this`. And the types of function pointers are derived from the types of the arguments and the return value. Obviously, the self-documenting bit of syntax `C::*` says that the function gets a `this` parameter of type `C`.

[33.2] How do I pass a pointer-to-member-function to a signal handler, X event callback, system call that starts a thread/task, etc?

FAQ: You don't. A member function can't be used without an object of the class, so the whole thing can't work. What you can do is write a non-member function wrapping your pointer-to-member-function call.

For example, thread creation callbacks usually have a `void*` argument. You could pass an object pointer in that argument to the callback (which has to be a non-member). The callback would then cast the `void*` down to the actual type and call the object's method.

Some functions, like `signal`, use callbacks without a `void*` argument or anything similar. In that case, you have no choice but save a pointer to the object in a global variable. The callback can get the object pointer from that global variable and call the method.

`static` member functions *can* be used in the contexts where a C callback is expected, if they are `extern "C"`. Although on most compilers it would probably work without `extern "C"`, the standard says it doesn't have to work.

FQA: The picture painted by the FAQ isn't very pretty, but the reality can get even worse - that is, more code to write. For example, you may want to call *any* method of an object - to select the method to call at run time, not compile time. *That* would really mean that you want to pass a pointer-to-member-function as a callback, which is what the question is all about. In the scenario in the FAQ, your problem is *passing the object pointer*, but there's actually *no pointer to a member function*.

Anyway, in this "full-blown" use case, passing just the object pointer via the `void*` argument is not enough. You'd have to wrap the two pointers (the object pointer and the function pointer) in a structure, pass a `void*` to that structure to your callback and unpack the structure in that callback. Simple, but quite verbose. Pretty much like implementing function calls in assembly.

This illustrates the fact that C++ is a very low-level language. When you have a Python object `obj` with a method `func`, and you want someone expecting a callback to call `obj.func()`, you can create the callback object with the expression `obj.func`. As simple as that. In some high-level languages doing this is equally easy and in some it's more verbose, but *never, ever* would you have to save a pointer to your object to a global variable (and worry about making it thread-local when relevant, etc.).

The problem is that in C++, there's no single concept of a "callable object" - instead, there are unrelated low-level mechanisms for calling functions. For example, non-member function pointers work differently from member function pointers. There are ubercompetent people out there who actually think they can get away with casting `void (C::*p)(int)` to type `void (*)(C*,int)`, because, you know, what we need is to pass `this` as the first parameter. This can work with many compilers, until you need to pass a pointer to a `virtual` function. Outsmarting compilers is usually dumb, especially C++ compilers. There really *is* more than one function call mechanism, and the different kinds of function pointer are not convertible - you have to implement adapters of varying degrees of clumsiness.

Regarding the `static`-members-as-callbacks issue: if your implementation uses different binary calling conventions for C functions and C++ `static` member functions, call the support and inform them that their developers consume mind-altering chemicals at work.

[33.3] Why do I keep getting compile errors (type mismatch) when I try to use a member function as an interrupt service routine?

FAQ: This is a special case of the previous question.

FQA: It is. So we'll use the opportunity to - surprise! - point out that handling interrupts in C++ can be done *just as well* as anything else. *Not much of a compliment*, really, but worth noting.

One of the problems with C++ is all the *unjustified* criticism. For example, some people will scream something like "What?! Handling interrupts in C++?!" - possibly followed by (false) claims about C++ being a "high-level language" (yeah, right) and maybe remarks about your mental health and stuff. These people don't know what they're talking about, and can make other people believe that C++ only looks like a bad thing if one doesn't know what he's talking about.

Clarifications: of course you shouldn't throw exceptions in interrupt handlers, or call `new`, etc. Of course doing the job in C would be better, but this isn't special to interrupts. Of course many C++ features can do more damage when you use them to talk to hardware than elsewhere. All I'm saying is that when you criticize something, you either accompany your claims with some reasoning or you have no chance to convince people (at least not the ones worth the effort).

[33.4] Why am I having trouble taking the address of a C++ function?

FAQ: Mmmm, you're trying to take the address in order to use it as a C function pointer, aren't you? Well, don't do that. And don't try to cast your way out of this, it won't work.

FQA: The FAQ didn't answer your question, did it? Instead, it assumed it knew what your problem was, and then answered a different question. Well, you could also ask your original question for a different reason. Specifically, you are already aware of the fact that `&C::f` has a different type than `&f`, but you don't know how to spell that type. So you're doing something semantically sensible, but you can't get the syntax right, because C++ has so much syntax making so little sense.

Well, I don't know the type of `&C::f` either, because it depends on the arguments; I only know it's something like `T1 (C::*)(T2,T3,T4)`. So here's a way to find out the type of an arbitrary C++ expression:

```
template<class T>
void show_type(const T&)
{
  int eat_flaming_death[-1];
}
void test_func()
{
  show_type(&C::f);
}
```

The compiler will then say something like `In void show_type<TheTypeYouWantedToFigureOut>(): arrays of negative size are not allowed`. In our case, `TheTypeYouWantedToFigureOut` will be substituted with the type of `&C::f`.

Here's the best part: there's a large "compile time assertions" movement promoting a plethora of arcane macros and templates which, like `show_type`, will cause the compiler to fail with an error message hopefully mentioning something related to the problem. This kind of thing is a *best practice* in C++. Isn't life amazing?

[33.5] How can I avoid syntax errors when calling a member function using a pointer-to-member-function?

FAQ: With a `typedef`, making the type name readable, and a `#define` macro, making the `((obj).*(func))` syntax readable. Ewww, macros are evil!

There were *hundreds* of postings to `comp.lang.c++` about this, says the FAQ. The layer of syntax proposed above could save the traffic.

FQA: The FAQ actually proposes to use the old and oh-so-evil C macros to cover up the brand new syntax introduced in C++. The FQA will avoid further comments on this advice, since the target is too easy.

The amazing part about the hundreds of messages is not the fact that people can't get the C++ syntax right. The amazing part is the fact that people *wanted to use* pointers to member functions, despite the fact that they are pretty useless. What you really need quite frequently is "delegates" or "functors" or "closures" - well, anything that represents *both* the code (a function/an expression/...) and the data (an object/bound local variables/...). C++ doesn't support these things very well. Perhaps the `comp.lang.c++` posters tried to implement something like this on top of C++ object and member function pointers. Maybe even something generic, with templates inheriting from abstract base classes involved. Yeah, that sounds quite like the favorite pass-time of C++ developers.

[33.6] How do I create and use an array of pointer-to-member-function?

FAQ: First, add a `typedef` and a macro. Then, use `FuncPtrType arr[] = {&C::f, &C::g, &C::h};`

FQA: Hey, why are we using an evil C array? Me not like this. How about:

```
std::vector<FuncPtrType> arr;
arr.push_back(&C::f);
arr.push_back(&C::g);
arr.push_back(&C::h);
```

There, it's much better now. Wait till you see the full type name of `arr` in a compiler error message.

[33.7] Can I convert a pointer-to-member-function to a `void*`?

FAQ: No, and if it seems to work on some platform, it doesn't make it legal.

FQA: Listen to the FAQ. This isn't just language lawyer talk - it *really* isn't going to work. Check out this article - it has a lot of material on C++ member function pointers. **WARNING:** this stuff can be used to scare little children.

The bottom line is that unlike a global function pointer, a member function pointer is not just the address of the first instruction of the function in most implementations, apparently with the exception of the compiler by Digital Mars (the company behind the D language). That compiler generates "thunk code" which handles the differences between various dispatching mechanisms (`virtual` vs. statically dispatched functions, different kinds of inheritance), and uses the address of that thunk code to represent member function pointers. Quote from the article about this implementation: "Why doesn't everyone else do it this way?"

So, really, don't cast these things to `void*` - you can't even sensibly cast them to non-member function pointers.

[33.8] Can I convert a pointer-to-function to a `void*`?

FAQ: Stop that. No. And don't tell me it worked for you. It's illegal.

FQA: C and C++ strongly separate between code and data (so do many languages and hardware processor implementations). A pointer to a data object is not the same as a pointer to a function. However, in the vast majority of implementations their sizes are going to be the same, so it's possible to convert a function pointer to a `void*`, and then somewhere else convert it back and call the function.

This violates the language rules. So does code assuming 2's complement integers and IEEE floating point. The chance of both kinds of code to actually fail on an interesting platform is low. Admittedly, the code-pointers-are-just-like-data-pointers assumption is less useful than the signed-numbers-can-be-divided-using-right-shift assumption, though. So typically one wouldn't do this kind of cast, after all.

[33.9] I need something like function-pointers, but with more flexibility and/or thread-safety; is there another way?

FAQ: A functionoid is what you need.

FQA: "Functionoid" rhymes with "marketroid", and is a term local to the FAQ, used instead of the standard term "functor".

[33.10] What the heck is a functionoid, and why would I use one?

FAQ: It means "a function on steroids", of course. The FAQ goes on to describe a class with a single `virtual` function called `doit`. An object of this class is basically just like a function except you can pass it arguments in its constructor, and it can keep state between calls without thread-unsafe global variables. The discussion is very lengthy and didactic, there are lots of examples. If you didn't say to yourself something like "Oh, yeah, that problem", you can follow the link to the real FAQ's answer to see what this is all about.

FQA: A "functionoid" (or functor, as it's normally called) is basically a manual emulation of closures. Closures can save all those little classes people create in order to have a function-pointer-plus-some-context. For example, the following code, which tries to use a higher-order function (`for_each`) in a language without closures (C++), is completely ridiculous:

```
struct Printer
{
  std::ostream& out;
  Printer(std::ostream& o) : out(o) {}
  template<class T>
  void operator()(const T& x) const { out<<x<<std::endl; }
};
void print_them(const std::vector<int>& them)
{
  std::for_each(them.begin(), them.end(), Printer(std::cout));
}
```

Clearly, a `for` loop wouldn't be nearly as bad. However, if we had closures, we could do something like this:

```
void print_them(const std::vector<int>& them)
{
  std::for_each(them.begin(), them.end(), lambda(x) { std::cout << x << std::endl; });
}
```

This would still be verbose because of smaller problems (`std::`, mentioning `them` twice), but at least the stupid `Printer` class is now replaced with code generated implicitly by the compiler.

Check out the monstrous boost lambda library designed to work around the lack of closures in C++ in a desperate attempt to make higher-level functions of the kind defined at `<algorithm>` not entirely useless. When I tried it, `gcc` wouldn't compile it without `-ftemplate-depth=40` (the default template nesting depth limit, 17, is not enough for this library), and I got *5 screens* of error messages from *a single line of code using the thing*. See also this thread, especially the part where they explain how `cout << _1 << endl` works but `cout << "\t" << _1 << endl` doesn't (to get there quick, search for "fails miserably", then continue to the reply).

[33.11] Can you make functionoids faster than normal function calls?

FAQ: Sure! Instead of `virtual` functions, use `inline` member functions and make the code using the "functionoid" a template.

FQA: This way the "functionoid" will compile slower than "normal function calls" though, not to mention the loss of flexibility at run time. The run time speed impact of the obsolete approach to inlining used by C++ is discussed here. The compile time speed impact of C++ templates is discussed here. The fact that it's *your* job to make sure things are inlined properly in this family of scenarios is one problem with the lack of closures in C++ discussed above.

Templates

This page is about C++ templates, one of the largest cannons to aim at your feet that the C++ arsenal has to offer. Templates solve the problems with C macros by creating 2 orders of magnitude more problems.

[35.1] What's the idea behind templates?

FAQ: A template describes how to build definitions (classes or functions) which are basically the same.

One application is type-safe containers; there are many, many more.

FQA: Let's get a bit more specific. The FAQ's answer is applicable to C macros, Lisp macros, ML functors, functions like `eval` found in many interpreted languages, OS code that generates assembly instructions used to handle interrupts at run time, and just plain code generation (writing programs that print source code). The purpose of all such devices is meta-programming - writing code that works with code, creating pieces of code which are "basically the same" (after all, they are built from the same rules), and yet have some interesting differences. The question is, how do we specify these rules and these differences?

The approach used in C++ templates is to use integral constants and types to represent the differences, and to use class & function definitions to represent the rules. The first decision prevents you from generating code dynamically, because the parameters can only be compile-time entities. The second decision prevents almost everything else, because you don't get to use a *programming language* to generate code - the only thing you can do is write code with some things factored out and made parameters. You can't do as simple and useful a set of "basically the same" classes as automatically generated class wrappers for remote procedure calls instead of normal "local" function calls (called "proxies and stubs" in COM terminology; there are many other terms). Even computing the factorial of an integer parameter is done using so much code abusing the language mechanisms that people with no useful work to do are *proud* of being able to accomplish this.

Beyond those fundamental limitations, templates follow the tradition of C++ features of interacting poorly with each other. Templates can't be compiled because they are not code - they are, well, templates from which code can be generated once you have the parameters, and *then* you can compile it. C++, like C, defines no way to locate the compiled code of a definition given its name. Consequently, template definitions are placed in `#include` files, and *recompiled in each translation unit each time they are instantiated*, even if the exact same instantiation is used in N other files. This problem is amplified by the tremendous complexity of the C++ grammar (the most complicated part of it is probably templates themselves), making this recompilation very slow. If your code doesn't compile, you get cryptic error messages. If it does compile, you might wonder what it means. That's where the interactions of the C++ type system (pointers, arrays, references, constants, literals...), function & operator overload resolution, function & class template specialization selection, built-in and user-defined implicit conversions, argument-dependent name look-up, namespaces, inheritance, dynamic binding and *other* things kick in. The sheer length of this list should be convincing: neither a human nor a program (say, an IDE) has a chance against this unprecedented syntactic power.

Poor support for meta-programming is not necessarily a very big deal, because you can do lots and lots of things without it. That is, unless you work in C++. For example, there are no built-in lists or dictionaries in C++; the standard library provides templates you can use, so you can recompile the definition of each kind of dictionary each time you use it in a source file. In fact, most of the code in the C++ standard library belongs to a conceptually and historically separate library called STL, which stands for "Standard Template Library". For example, that's where `std::vector`, which the FAQ recommends to use instead of the evil C arrays, comes from.

If you use C++, chances are that you're going to deal a lot with its obscure meta-programming facilities.

[35.2] What's the syntax / semantics for a "class template"?

FAQ: You add the parameters before the definition of your class, as in `template<typename T> class Array { ... };`, and then you can use the parameters in the definition of the class, as in `T arr[N];`, and then you can use your class by substituting the parameters as in `Array<int>` (the FAQ gives a code listing instead of using words; its example is as simple as that).

FQA: Wow, that sounds easy! Too bad it's wrong, in two ways.

First, things will not follow this straight-forward model - the FAQ itself discusses a couple of cases, like the need to resort to the `typename` keyword and the look-up of "nondependent names". All such cases illustrate that you *can't* take the definition of a class, parameterize some things making it a template, and expect it to just work - it's way more tricky.

Second, what about the cases where nobody would *ever* write the classes generated from templates manually, but templates are still used because that's the only meta-programming facility offered by C++? Of course there are also C macros, which have some limitations templates don't have (and vice versa), and at least compile fast. But in the C++ community, macros are considered the most evil feature ever, and using them is treated as a sin somewhere between speeding and blasphemy. Anyway, a majority of all uses of templates "beyond type-safe containers" actually fall in this second category. Let's look at the previously mentioned compile-time factorial example, which is *trivial* compared to nifty stuff like type lists:

```
template<int N>
struct Factorial
{
  enum { value = N*Factorial<N-1>::value };
};
template<>
struct Factorial<0>
{
  enum { value = 1 };
};
```

This code generates N+1 classes in order to compute the factorial of N. Talk about "the syntax / semantics of class templates". This is equally disturbing to humans - because it makes so little sense - and to compilers, which internally represent each class using a sizable data structure. This compile time computation technique tends to take a lot of compile time. When your only tool is a hammer template, not only does every problem look like a nail - you also hammer it with 100 hammers.

Oh, and when you want to use a template, be kind with the C++ lexer (the part of the compiler converting text to "tokens" so that the parser can check whether they make meaningful statements). Code like `vector<vector<int>>`, which tries to declare an inefficient implementation of a 2D array, won't compile - you need a space before the two > characters. Their concatenation looks just like the right bitwise shift operator. This is one of the more harmless, albeit confusing, awkward interactions between C++ features.

[35.3] What's the syntax / semantics for a "function template"?

FAQ: Pretty similar to class templates, plus you can usually omit the template parameters - the compiler will figure them out from the function arguments. For instance, you can write a `swap` function for swapping two values of any type, be it integers, strings, sets or file systems (yes, the FAQ *actually mentions* swapping some `FileSystem` objects).

By the way, an instantiation of a "function template" is called "template function".

FQA: In addition to all the problems with class templates, we are now engaged in a battle of wits with the oh-so-smart compiler figuring out template parameters from function arguments. Or not. For example, `std::max(x,5)` compiles when x is a `int`, but fails to compile when it's a `float` or a `short`. You see, the point of templates is to make "algorithms" work with values of many different types, facilitating "code reuse" (the fact that `x>5?x:5` is less code than `std::max(x,(short)5)` doesn't mean you don't want to *reuse code*, does it?).

For instance, you can reuse `std::swap` to swap a couple of file systems. All you have to do is implement a class `FileSystem` with a default constructor creating a new empty disk partition. The copy constructor will copy all the files from a given `FileSystem` to a newly created partition, the destructor will wipe it out, and `operator=` will do the latter followed by the former. To handle errors, use exceptions. You might get a few extra disk partitions created and destroyed, especially if you pass `FileSystem` objects around too much in the code, but that's a small price to pay for reusing the 3 lines of code in `std::swap`. And a "commercial-grade" compiler can even eliminate some of those copies!

The note about "function templates" and "template functions" is very useful. Too bad there are people out there that confuse the two. Be careful with C++ terminology. For example, don't confuse "object files" (compiled code) with "objects" (which belong to a class), which in turn shouldn't be confused with "instantiation" of class templates (substituting template parameters is called "instantiation", the result of this process is also "instantiation", and creating objects is called "construction"). The good (or bad) news is that the terminology is the easy part.

[35.4] How do I explicitly select which version of a function template should get called?

FAQ: Most of the time you don't need to do it - the compiler will guess. It uses arguments to guess, so when your function has none, use `f<int>();`. Sometimes you want to force the compiler to choose a different type than it would choose by default - for example, `g(45)` will instantiate `g<int>`, while you want `g<long>`. You can force the compiler to call `g<long>` with explicit instantiation as in `g<long>(45);` or type conversion as in `g(45L);`.

FQA: There is a good reason to avoid all kinds of "clever" behavior which gets in your way when you try to figure out what a program actually does (it doesn't always do what the author thought it would do). Overloading and template specialization are one kind of this behavior - go figure which function is actually called. But let's assume for a moment that it's not a problem, and that the important thing is to write code expressing the author's intent "clearly" in the sense that it's not cluttered with "low-level" details like which "version" of a function is called.

In that case, the picture presented by the FAQ looks fair - you only need to explicitly specify parameters when the compiler has no information to guess them itself, or when you don't like its guess. The truth is more complicated, because sometimes a function has more than one argument, and the template defines constraints on these arguments. That's what happens with `std::max(x,5)` when x is a `float`. The template wants two arguments of *the same* type, and the compiler thinks that "5" is of type `int`. So even though the intent seems clear (you probably want 5 to be treated as a number of the same type as that of x), the compiler can't make a decision. So you have to interfere in these cases, and choose between two almost equally unreadable alternatives, explicit instantiation or explicit type conversion.

Out of the two options, explicit instantiation, the syntax defined by C++ (as opposed to type conversion which is inherited from C) is typically worse. First, this way the code using a function forces it to be implemented as a template, although it doesn't really care, and makes it harder to get rid of the pesky templates when you feel like it. And second, when the need to interfere and disambiguate arises *inside another template*, you may have to use the following syntax, hideous even by C++ standards:

```
a.template f<long>(45);
```

While C++ features generally interact poorly with each other, templates set the record by interacting poorly with themselves.

[35.5] What is a "parameterized type"?

FAQ: A way to say "class templates".

FQA: Hmm, why do we need two ways of saying this?

[35.6] What is "genericity"?

FAQ: A way to say "class templates". Don't confuse with "generality".

FQA: Hmm, why do we need three ways of saying this? Apparently C++ promoters consider templates an excellent and unique feature, despite their obscurity and the availability of much better meta-programming facilities in many languages. The many different synonyms are probably needed to illuminate the killer feature from many different directions.

[35.7] My template function does something special when the template type `T` is `int` or `std::string`; how do I write my template so it uses the special code when `T` is one of those specific types?

FAQ: First, make sure it's a good thing to do in your case. Generally, it is when the "observable behavior" in the special version you want to add is identical to the general case - otherwise, you're not helping your users. If the special case is "consistent" with the generic case, you can do it as in `template<> void foo<int>() { ... }`.

FQA: "Observable behavior" means different things to different people. In particular, many C++ users tend to *observe* performance (if you don't care about performance, but you still use a language that won't detect run time violations of language rules like out-of-bounds array indexes, you're probably wasting your time).

Consequently, the specialization of `vector<bool>` to be space-efficient (by storing bits instead of bytes) at the cost of speed (individual bits are harder to access than whole bytes) found in some STL versions is not a very good idea, because it ultimately confuses the performance-aware user. If the user wants a vector of bits, the user can implement a vector of bits, or STL could supply one, but it's very inconvenient when you can't have a simple mental model describing what `vector` really means.

In addition, specialization is actually a pretty dangerous trap because it is your responsibility to make sure that all specializations are visible to the compiler (`#included`) at each point where a template is used. If that's not the case, you can get too kinds of `vector<bool>` instantiated in your program, triggering "undefined behavior" (typically you'll pass a vector of the first kind to a function compiled to work with vectors of the second kind and crash). So specializing *others' templates* is very likely to lead to a disaster (because you can't make sure that your specializations are visible in code you didn't write), and libraries which actually *assume* that you'll specialize templates they define are best avoided.

If you really want to use specialization, take into account that function templates don't support partial specialization ("I want a special version for all types which are vectors of any T"), only class templates do (template functions support *overloading*, which follows different rules). One workaround is to implement a single function template working with any T and delegate the call to a static method of a template class, as in:

```
template<class T>
void f(const T& x)
{
  FImpl<T>::f(x);
}
```

This way, the class `FImpl` can be defined using partial specialization. All these layers of cryptic syntax could be considered tolerable if they ultimately were the only way to accomplish something really useful, and you could forget about them once you were done. But it's actually very easy to program *without* these complications, and almost always these complications get you nothing except for reducing maintainability, and you get to see them each time you have a compilation error deep down a chain of templates delegating trivial work to each other, and debugging run time errors becomes a real nightmare.

Basically you can choose simple interfaces and simple implementations, or C++-style cryptic interfaces and cryptic implementations. It's a trade-off.

[35.8] Huh? Can you provide an example of template specialization that doesn't use `foo` and `bar`?

FAQ: For instance, you can "stringify" values of different types using a template. The generic version boils down to `ostringstream out; out << x`. But you might want to define specializations to handle types where the `ostream` output operator doesn't do what you like (you can set the output precision of floating point numbers, etc.)

FQA: This means that all values of the same type will have to be formatted identically. For example, all integers will be printed using decimal digits. If you prefer hexadecimal, you can define `class HexInt` (a template, of course, so that it can handle all the different integral types, including user-defined ones). Then you can use `stringify(HexInt<int>(x))`. You might need a partial specialization of `stringify` for `HexInt<T>` (see previous FAQ). To save the trouble of explicitly passing the template parameters to `HexInt`, use a creator function template `HexInt<T> hexint(const T&)` - the compiler will figure T out from `stringify(hexint(x))`. Specifying a number of leading zeros (as in the format string `"%08x"`) using advanced C++ type-based techniques is left as an exercise to the reader.

We've done quite some work indeed in order to print an integer. Time to relax and let the compiler concentrate while it cleverly figures out all the things we want it to figure out. You usually print stuff for debugging, so the long build time may be annoying, but it's sure better than using a visual debugger where you get to see the line noise generated from all those other templates.

In the meanwhile, it is quite likely that whatever your *real* job was, someone else (probably some kind of competitor) has already done it. But did the result contain a mature, generic and efficient printing infrastructure with really, really minor usability and maintainability problems? Most certainly it didn't. Which is why C++ template specialization is your friend.

[35.9] But most of the code in my template function is the same; is there some way to get the benefits of template specialization without duplicating all that source code?

FAQ: You can factor out the common code and only specialize a helper function called from the common code. Two screens of source code are attached for illustration.

FQA: Factoring out common code and helper functions are indeed very useful (which is why you can do that kind of thing with virtually every programming language). The existence of this question in the FAQ seems to indicate that people get unbelievably confused by templates, and lose hope that anything useful they know is applicable to them. Which is not that much of an exaggeration.

[35.10] All those templates and template specializations must slow down my program, right?

FAQ: You guessed wrong. Maybe the compilation will become "slightly" slower. But the compiler ends up figuring out the types of everything, and then doing all the usual nifty C++ optimizations.

FQA: You guessed right. And the compilation will become intolerably slow. It's not like the FAQ is lying - it's just talking about the state of affairs in theory, where C++ belongs. The "slight" slowdown of compilation is not even worth discussing: everything is "slight" if you have lots of time on your hands. Just try to build a C program and a *modern* C++ program full of templates and compare the time it took. As to execution time, there are practical problems making programs generated from templates slow compared to the hand-written alternatives.

First, the compiler generates the same code over and over again. Sometimes the linker throws away the extra copies and sometimes it doesn't, and the size of your program increases. In particular, the linker doesn't have a chance to throw away the functions which are identical at the *assembly* level, but not at the *source code* level (think about `vector<int>` and `vector<void*>`). It is possible to implement templates in a way avoiding these problems (by using the same implementation for all specializations yielding the same assembly code). It is very tedious and almost never done. Two identical functions almost always take more time to execute than a single function called twice, which has to do with instruction caches - a useful gadget frequently overlooked by many people who care about "theoretical efficiency" without actually *measuring* performance.

Second, when people work with templates, they use *types* - their only hammer - for saying almost everything (consider the `HexInt` class from the previous FAQ). More specifically, they wrap simple values of built-in types in user-defined types - classes. The type is used to select the right specialization and what-not - in fact it's used to specify what to do. An extreme example is the boost lambda library - it creates structures representing entire *functions*, with a sub-structure representing addition, a sub-structure representing the constant 1, etc.

Now, "theoretical performance fans" may think that all of these structures get optimized out by the clever compiler. In practice, that's almost always close to impossible to do because of the so-called pointer aliasing problem. When you have a local variable x, it's clear that nobody can change it but the code of the function, so the compiler can do lots of things with x, like stuffing it into a register or even completely optimizing it out. But once you push x into a structure, it's hard to see where it's modified - go figure who has a pointer to that structure, especially if you pass the object to a separately compiled function. So the compiler has to allocate a memory slot for x and make sure the memory cell gets updated when x is modified and that the memory cell gets read when there's a chance that it could have been changed by someone else. Code working with templates and relying on types to do compile time dispatching ends up doing lots of memory load/store operations at *run time* since the types don't really go away.

And anyway, there's such a huge amount of scenarios to take care of to optimize complicated template-based code well that compiler writers rarely bother. They are lucky if they get the *parsing* right. Even that is unlikely - when you are porting from one compiler to another, chances are that most of your compatibility problems will come from the semantics of the code using templates.

[35.11] So templates are overloading, right?

FAQ: They are in the sense that they are inspected when the compiler resolves names (figures out the version of `f` that should be called by `f(x)`). They are not in the sense that the rules are different. Specifically, there's the SFINAE (Substitution Failure Is Not An Error) rule: the argument types have to match exactly for a template to be considered in overload resolution. If they don't, the compiler won't try to apply conversions the way it would with a function - instead it will discard the template.

FQA: Aside from the fact that the acronym "SFINAE" interpreted literally doesn't seem to describe what it's supposed to describe, this sounds *just a little bit* too easy. For example, what does "exact match" mean? Let's have a look at a real life example taken from the GNU implementation of the C++ standard library. I tried to make this short, but there are about **5** distinct stupid things involved, so it was hard. If you get tired in the middle of this and stop, it's probably an indication that you *do* get the main point - that things are actually very complicated in this department and that these complications are best avoided.

Stupid thing #1: Once upon a time, `vector<int>::iterator` was a plain old `typedef` for `int*` in the GNU STL. Of course this is *very, very* dangerous: people might use `std::vector` as if it were just an array of objects - a serious abstraction violation, you could get arrested for that in some jurisdictions. At the beginning of the 21st century, the guys behind GNU STL decided to "fix" this by creating a class template called `__normal_iterator`. This template serves as a "strict typedef" - it wraps any existing iterator type, such as `int*`, and delegates all operations to the existing type, but can not be converted to `int*`. This has many important benefits over the previous implementation, for example, much longer error messages.

Stupid thing #2: As you may know, there are two kinds of iterators defined by STL containers: `iterator` and `const_iterator` because of inherent problems with const: `const iterator` is *not at all* the same as `const_iterator`. If STL wanted to be consistent, it would also define `volatile_iterator` and `const_volatile_iterator`, and then nobody would even look at STL, which wouldn't necessarily be bad. But they didn't. So now you also need two kinds of "normal iterators" - for `int*` and `const int*`.

Stupid thing #3: *Of course* the GNU STL guys didn't want to define a `__normal_const_iterator` - after all, making conversion hard for *you* is an excellent thing, but making it hard for *them* is a completely different thing. Instead, they decided to support automatic conversion between different instantiations of `__normal_iterator`, by - of course - delegating the conversion to the wrapped types (templates normally delegate all useful work to someone else; their job is obfuscation). This way, you can compare const and non-const iterators using the same operator, having this elegant prototype:

```
template<typename _IteratorL, typename _IteratorR, typename _Container>
inline bool
operator>(const __normal_iterator<_IteratorL, _Container>& __lhs,
          const __normal_iterator<_IteratorR, _Container>& __rhs);
```

Stupid thing #4: STL provides another, *seemingly unrelated* service to its users. It defines global relational operators which work on arguments of any type. How can they compare objects without knowing anything about them? At this point you can probably guess the answer - of course, they delegate the work to someone else, this time to *existing* relational operators, using interesting identities such as `(a>b) == !(a<=b)`. This way, you can define only 2 relational operators, and get the rest "for free".

Stupid thing #5: Except when you can't. This is where our subject, overload resolution and templates, kicks in. Remember the rule about only considering templates when the argument types match the prototype *exactly*? Well, when you compare "normal iterators" of the same type (for example, both wrapping `int*`), the beautiful prototype above matches the arguments "exactly". So do the "generic" relational operators. Oops, we have ambiguous overloading! The "solution" is to define a *third* overload, matching the types *even more exactly* by using a single template parameter `_Iterator` instead of two which can possibly differ. Why is this situation considered "unambiguous"? Frankly, beyond the basic intuition saying that you must show the compiler a type pattern as similar to your arguments as possible, I don't know. That's why I didn't list the Stupid thing #6. But the first 5 make me feel that in this case, *ignorance is bliss*.

Apparently this situation looks discouraging even from inside the C++ universe, as indicated by the following *rather sad* comment found in one of the header files of the GNU STL. You can probably decipher it, unless the 5 stupid things above have already faded from your memory:

```
// Note: In what follows, the left- and right-hand-side iterators are
// allowed to vary in types (conceptually in cv-qualification) so that
// comparison between cv-qualified and non-cv-qualified iterators be
// valid.  However, the greedy and unfriendly operators in std::rel_ops
// will make overload resolution ambiguous (when in scope) if we don't
// provide overloads whose operands are of the same type.  Can someone
// remind me what generic programming is about? -- Gaby
```

This could be amusing (an implementor of the standard library of a language complaining about this language in files delivered to users and all) if it weren't so mind-numbing. People who think they are better at C++ than the GNU STL authors are welcome to waste their entire life chasing and "solving" problems with overload resolution, template specialization or whatever its name is. For the rest, trying to avoid templates & overloading sounds like a good advice, which can be followed to an extent even if you are forced to use C++.

[35.12] Why can't I separate the definition of my templates class from its declaration and put it inside a .cpp file?

FAQ: "Accept these facts", says the FAQ - templates are not code, just a recipe for generating code given parameters; in order to compile this code, it must first be generated, which takes knowing both the template definition and the definitions of the parameters which are passed to it; and the compiler doesn't know anything about code outside of a file when compiling the file (which is called "separate compilation").

So you have to place template definitions in header files, or else the compiler won't get a chance to see all the definitions it needs at the same time. Experts should calm down - yes, it's oversimplified; if you know it is, you don't need this answer anyway.

FQA: There are two problems with placing template definitions in header files which may bother you: you get to recompile them each time the file is included, and you disclose your source code to the user. Let's concentrate on the first problem. The second is minor anyway because source code of templates isn't necessarily easier to understand than disassembled object code. As to the first problem - to get an idea about its magnitude, consider the fact that an iostream-based "hello, world" program requires the GNU C++ compiler to parse **718K** (!!) bytes. And contrary to the claims in the FAQ, it turns out that the need to make the source code of templates available to the compiler is not the only reason we have this problem.

Suppose a C++ compiler could use some rules to locate compiled definitions of classes given their names, for example "`std::vector<int>` is always located at the file `$OUTPUT_ROOT/templates/std/vector_int.o`" and so on. This way, if you used `vector<int>` and `vector<double>`, you'd have to compile `vector` twice, but if you used `vector<int>` twice, the compiler could avoid recompilation. That would make sense since you'd only compile *different* classes each time you compile the template.

Unfortunately, this can't work in C++. That's because the compiler can't *parse* `std::vector<int>` without parsing the entire preprocessed output generated by `#include <vector>`. That parsing, which *has* to be done over and over again for each compiled source file, takes most of the compilation time. Generating the code of `std::vector<int>` several times is the small part of the problem, and most compiler writers don't bother to solve it, since you'd still have the parsing bottleneck.

The basic problem inherited from C is that the compiler can't look up definitions. Instead, you have to arrange `#include` files so that the preprocessor can copy-and-paste definitions into a single huge bulk containing everything relevant (as well as many irrelevant things) for the compilation of your source file. C still compiles fast because its grammar is simple. Many newer languages define not only the concept of a "class", but also rules to help the compiler locate definitions instead of parsing them over and over again. C++ programmers enjoy the worst of both worlds.

[35.13] How can I avoid linker errors with my template functions?

FAQ: You probably didn't make the definition of a template available to your compiler at the point where a template is used (did you implement a template in a .cpp file?). There are three solutions:

- Move the definition to the .h file (which may increase the size of your compiled code, unless the compiler is "smart enough").
- Add explicit instantiations in your .cpp file. For example, `template void foo<int>();` will cause the compiler to generate the code of `foo<int>`, and the linker will find it.
- `#include` the .cpp file defining the template at the .cpp file using the template. If it feels weird, live with it or read the previous FAQ.

FQA: These "solutions" create new problems:

- **Moving code to .h file**: even if the code size doesn't increase, the compilation time will - you are going to recompile

your templates from scratch each time they are used, and templates are one of the hardest parts of C++ to compile. It's nice that the FAQ at least acknowledges that your code size *might* increase though - in practice it most certainly *will*. Too bad the FAQ didn't mention it in the question about the speed of templates - replicated code means more time spent in fetching code into instruction caches.
- **Explicit instantiations**: this is one big step back to C macros. Suppose you have a parameterized preprocessor macro which expands to a definition of a container class. One difference between this macro and an equivalent template is that if someone *uses* one such class, that someone must also make sure that the macro is actually *expanded* somewhere with the appropriate parameters - or you must provide these expansions. Otherwise, there will be linker errors. With templates, the compiler is supposed to generate the needed instantiations transparently - except it's almost impossible to accomplish. Along comes this "solution", moving us back to square one.
- **Including the definition at the point of usage**: this shares the problems of the first two options. If you use templates in .h files, you may have to include the .cpp files defining them in these .h files, getting the problems of option 1. But the benefit of option 1 - transparency - is gone: quite similarly to option 2, it is now your job to make the definition of a template available wherever the template is used.

[35.14] How does the C++ keyword `export` help with template linker errors?

FAQ: It's "designed" to eliminate the need to make the definition of a template available to the compiler at the point of usage. Currently, there's only one compiler supporting it. The keyword's future is "unknown".

An advice for futuristic programmers follows. It shows a way to make code compatible with both compilers that support `export` and those that don't using - guess what? - the wonders of the evil C preprocessor. Among other things, the FAQ advises to `#define export` under certain conditions.

FQA: "How does it help", is that what you want to know? OK then. The `export` keyword helps with template linker errors just the way a song about peace helps to stop a bullet penetrating a foot. It helps just like a keyword `findbugs` telling the compiler to find and report all the bugs in a piece of code would help you with bugs.

The rest of C++ makes this keyword impossible to support in any useful way that would actually yield faster compilation compared to the case when template definitions are included at header files. That's why most compilers don't bother to support it, and that's why the future of the keyword is "unknown": it's useless.

If you spot someone following the FAQ's advice (`#ifdef EXTRATERRESTRIAL_COMPILER` and all that), call an ambulance. Warning: the patient has likely reached a very agitated state and might escape before the people qualified to deal with the situation arrive. Try to occupy the patient's mind with a discussion about the fact that `#defining` keywords is illegal C++. Propose to consult your lawyer. Try to "design" a couple of keywords together (look up synonyms in a dictionary, imagine them printed in popular fonts, stuff like that). Improvise. It's gonna be over soon.

[35.15] How can I avoid linker errors with my template classes?

FAQ: It's just like errors with template functions, which were explained in the previous answers.

FQA: Yep, it's about the same.

[35.16] Why do I get linker errors when I use template friends?

FAQ: If you have a class template `C` declaring a friend like `Foo<T> f()`, the compiler assumes that there's a global function `f()` returning `Foo<T>`. You probably meant a different thing, namely - there's a function template `template<class T> Foo<T> f()`, and you want its instantiation `f<T>()` to be a friend of your instantiation `C<T>`.

There are two ways around this:

- Declare the function template before the definition of the class, and add `<>` to the friend declaration, as in `friend Foo<T> f<>();`
- Define the friend function template inside the body of the class.

FQA: In both solutions the syntax is unrelated to the semantics to an extent remarkable even for C++.

Why does `<>` mean that we are talking about an instantiation of a template function? Why not use a keyword (like, just an example off the top of the head, the `template` keyword) to say that?! The C++ way is uncompromisingly ugly, especially in the example mentioned by the FAQ itself: `operator<< <>(...)`.

And even when you don't have a problem with placing the definition in a header file, the seemingly cleaner second way is actually *more* cryptic. This breaks one of the *very few* things in C++ you can normally count on: that a declaration of a function or a type looks just like a definition without the body. Here, we change the meaning of the prototype by adding a body: instead of declaring a function, we now declared and defined a function template.

Last but not least, the very fact that this problem *exists* is an indication of a readability problem with the C++ grammar. How many people would guess that the original declaration refers to a function and not a function template?

How is one supposed to navigate through this swamp of arbitrary syntax? *Of course* one shouldn't expect the kind of readability you get with a natural language from a programming language. *Of course* any formal language will behave "counter-intuitively" at times. But people *do* deal with formal languages quite successfully, when it is possible to keep a reasonably compact model of the key rules in one's mind. In these cases, even if you bump into a behavior which doesn't make sense at the first glance, you can think again and - "Of course, of course, I know what it's doing!". Do you feel that you understand what a C++ compiler is actually doing? Neither do most C++ users out there.

[35.17] How can any human hope to understand these overly verbose template-based error messages?

FAQ: There's a "free tool" converting compiler error messages to more human-readable ones. It works with many compilers.

An example follows, having a snippet `[STL Decryptor: Suppressed 1 more STL standard header message]` in it.

FQA: Oh *really*? *Any* "template-based" error messages? Hmm, why didn't the compiler writers produce clean error messages in the first place if a single tool can clean up all the mess created by many different compilers? These people must be quite lazy and/or stupid. Or are they?

Actually, no, they are not. The error messages are cryptic because templates are cryptic, and most compilers can't really do much better than they do today.

The tool mentioned (*without the name*) and linked to by the FAQ is called `STLFilt`. That's why the FAQ doesn't mention the name. That's why its output does mention STL. That's why it works at all - you can't improve generic template error messages, but you can filter STL-related messages if you know how STL is implemented in each specific case.

We need a couple more tools, like `STLCompilationTimeReducer` and `STLDebugInformationBeautifier`, and we're all set. Yet another proof that generic meta-programming facilities are a better way to implement containers than build them into a language.

If you wish to implement a template library, don't forget to implement a tool filtering the error messages your users will get, as well as the other cool tools, for all the flavors of compilers out there.

[35.18] Why am I getting errors when my template-derived-class uses a nested type it inherits from its template-base-class?

FAQ: This can hurt, sit down. The compiler doesn't look for "non-dependent" names (ones that don't mention the template parameters) in "dependent" base classes. So if you inherited a nested class or typedef `A` from your base class `B<T>`, you can only access it using a dependent name, like `B<T>::A`, but you can't use a non-dependent name, like plain `A`.

And you'll have to prefix that "dependent" name with the `typename` keyword. That's because the compiler doesn't know that `B<T>::A` is a type (think about two specializations, one defining a nested class `A` and one defining a global variable `A`).

FQA: This illustrates two generic problems with the C++ grammar.

First, class templates are *not* just parameterized class definitions, because the crazy C++ name look-up gets crazier when you are inside a template. So don't assume you can take a C++ class definition, factor out a bunch of parameters and get a working template definition.

Second, telling a C++ type name from a C++ object/function name is insanely complicated. This interacts badly with templates, constructors, and everything else.

As to the possible reactions of users the FAQ attempts to anticipate: while *running away* may be justified, *sitting down* is probably not. Good developers tend to *test their code*, so even if the compiler didn't spit an error message and did the wrong thing silently (for instance, used a type called `A` from the global namespace), a test will find the error. Stupid compiler behavior only feels like pain for people who think that the extremely slow C++ compilers spend their time in finding all their bugs, and don't bother to test the result. Those people should relax and save their tears for the C++ *run-time* errors.

[35.19] Why am I getting errors when my template-derived-class uses a

member it inherits from its template-base-class?

FAQ: The reasons are identical to those in the previous FAQ. But the workarounds are *different* - convert `f()` to `this->f()` or add the statement `using B<T>::f()` to your class definition. Using the fully qualified name like it was done in the previous FAQ (`B<T>::f()`) will also work, except when the function is virtual, in which case the compiler will use static binding, not dynamic binding.

FQA: Yep, the case with member functions is similar to the case with nested types, with the additional bonus of interacting badly with virtual functions.

The FAQ has a hilarious comment in the spirit of "this doesn't mean that template inheritance doesn't work - but the name look-up works differently". And that's C++ for you: it's not like you can't write code, it's just that you can't tell for sure what any particular name or some other part of it means.

[35.20] Can the previous problem hurt me silently? Is it possible that the compiler will silently generate the wrong code?

FAQ: Yes - the compiler might call a function or use a type from the global namespace instead of what you meant, for example.

FQA: It's not as horrible that a language can silently misinterpret what you mean as it may sound. Any language will do this to the creative but imprecise human mind (formal languages aside, people frequently misunderstand each other).

With formal languages, you can form a relatively simple model which will help you understand these problems. It's the same with C++ except for the "simple" part, so C++ and you will misunderstand each other pretty frequently.

And you can also test your code by creating programs that check if it does what you want it to do in a bunch of cases. It's the same with C++ except that it compiles forever and you have to write notable amounts of code to implement the simplest test, so C++ code ends up being tested pretty rarely.

FQA errors

This page lists the factual errors/inaccuracies in the C++ FQA. If you find one, please send me e-mail. If you are right, I'll publish your correction, either giving you the credit or anonymously, according to your choice.

By "factual errors", I mean statements which can be proved wrong or refuted by a practically feasible test.

Positive examples: if I say that C++ compilers may generate slower code from C source than C compilers unless exception support is off, and in fact no commercially significant compiler does that, I'm wrong (don't bother with this one - I already checked it). If I say that the C++ grammar is undecidable, and you formally prove it's decidable, I'm wrong (advice: don't bother with this one, too).

Negative examples: if I say templates are mostly applicable to containers, and you know many other ways to use templates, it's really a qualitative argument. You call this "uses", I call it "abuses". It's like arguing whether `sed` is applicable to numerical computing: it's a matter of common sense, and we'll get nowhere - there's no formal definition of a Turing tarpit. If you inform me that I'm "inconsistent" because "sometimes my problem with C++ is that it's too low-level for high-level work and sometimes I think it's too high-level for low-level work", the message will land in the bit bucket, too. I think that doing both low-level and high-level work in C++ is suboptimal (I tried both); you are free to call it "inconsistency". If you think the fact that C++ is a "superset" of C means that C++ can't be inferior to C, go visit Chernobyl and buy yourself a radioactive cat with 7 legs and a wing (codenamed "cat++"), but don't expect me to agree.

Sometimes base class constructors must have information on the actual object type

Eugene Toder: The FQA answer about the dispatching of virtual functions in constructors is based on the following statement: *Base::Base doesn't know that it's called in order to ultimately initialize a Derived object*. This is wrong, since virtual inheritance can't work that way. In particular, AFAIK, `cfront`, the first C++ compiler, always passed constructors a parameter telling whether it initializes a base class object or a derived class object, no matter what kind of inheritance was involved.

Yossi: This is an error by itself, and it's also inconsistent with other information in the FQA. Specifically, the FQA mentions (following the FAQ) that with virtual inheritance, the programmer must directly initialize virtual base class objects in the derived classes, even if they are not its "immediate" base classes (that is, it inherits them indirectly). But what if someone derives another class, `Derived2`, from a class `Derived` with virtual base classes? It's *still* up to the programmer to initialize the virtual base class objects in `Derived2` - but `Derived` already contains code that does this (and maybe *other* base classes contain such code, which is our problem in the first place). So we *don't* want to use the virtual base class initialization code in `Derived`, but we *do* want to use the other initialization code in `Derived`. Therefore, the constructor of `Derived` must know whether it ultimately initializes a `Derived` object or an object of some child class, such as `Derived2`, in order to conditionally execute some of the initialization code (or the constructor code must be generated twice, which is the same in this context).

I think that the FQA answer still has its value in the sense that it may actually be more intuitive to C++ programmers. That's because the FAQ answer basically says, "C++ prevents you from a *potential* (not certain) error of accessing uninitialized members of a derived class, by silently doing something different than what you thought it would". This is a very special fact which has to be memorized; normally C++ *doesn't* try to prevent access to uninitialized data. The FQA answer says, "C++ does the thing naturally and efficiently following from the underlying implementation: to set up `vptr` to point to the correct `vtable`, you'd have to spend cycles (a tiny amount of them, but C++ is frequently very conservative about run time cost)". Arguably, this is more consistent with the rest of C++ and is easier to remember. But of course it doesn't make the erroneous statement in the FQA correct.

C++ has `bool`, a built-in type not in C

From a Usenet posting (in comp.lang.c++.moderated): You said "C++ doesn't add any built-in types to C." What about `bool`?

Yossi: This statement is false (wait, that's not good, I mean the one quoted in the posting). The FQA does mention that C++ adds built-in types which are essentially new kinds of pointers (references, pointers to members), but it doesn't say this in Defective C++ where the false assertion appears, and it fails to mention `bool` in this context anywhere.

Unlike the previous item, this error doesn't invalidate the reasoning in the text where it appears. Specifically, the context of the erroneous statement is the discussion of *high-level* built-in types, which don't map directly to C types the way `bool` does, primarily because their sizes are not known at compile time.

`#define private public` is not enough to cancel C++ encapsulation

As Andreas Krey pointed out, you'd also need `#define protected public` and `#define class struct`. The latter will fail to work with code like `template<class T> class X {...}`, because the keyword `struct` can not be used in template parameter lists. There may be other problems related to interactions between access control and name look-up.

Another thing I failed to mention in the original context: re-#defining keywords isn't legal C++. It will only work in implementations where preprocessing is a separate pass, which is unaware of the keywords and treats them as identifiers. I've never worked with an implementation doing it differently, but `#define private public` is still illegal C++. It's only useful for debugging (to print private data without changing header files and recompiling for hours), and to illustrate that C++ encapsulation is useless for security (the latter was my original point).

The special case when N-dimensional arrays can be allocated dynamically with `new`

When all array dimensions except for the first are known at compile time, you can allocate arrays dynamically with `new`. That's because C++ arrays are flat sequences of objects of the same type. In our case, those objects are (N-1)-dimensional arrays. If any of those N-1 dimensions weren't known at compile time, the (N-1)-dimensional array wouldn't be possible to describe with a C++ type. But not knowing the first dimension until run time doesn't create this problem.

Joe Zbiciak found that answer 16.16 fails to mention this. Answer 16.20 does, but 16.16 is still incorrect because it says you only have 2 ways to allocate N-dimensional arrays dynamically. However, in our special case, you have a third way.

Conversions between code and data pointers are common

The FQA says that conversions between function pointers and void pointers are rare (independently of being illegal, though supported on many machines). Patrick Walton counters:

"Actually, this is done all the time for dynamic linking, in `dlsym` on POSIX platforms (returns a `void*`, even for code pointers) and `GetProcAddress` on Windows (returns a `void (*)()`, even for data pointers). Interesting that both platforms violate the standard in opposite ways. In the case of POSIX, there's a lot of pain involved in making this required interface work on systems where function pointers are longer than data pointers."

I wonder why neither Windows nor POSIX defined two functions, one returning a `void*` and one returning a `void (*)()`.