My C++ Feature Guide, and What I Think Should Be Preferred

By Aaron Fritz (1/7/2016)

Things which, when applicable, should always be used to follow modern C++ style, roughly ordered by my view of their importance:

- **std::string**, because char* is error-prone, lots of busy work for strcat/strcpy, and doesn't allow for nice things like the concatenation operator. All allocations and de-allocations are taken care of, too, and you can even get a const char* from a std::string via "c_str()" if necessary. There is always the argument about performance, but what is there that needs to be done ten million times a second that requires character manipulation? Use char* in that situation, but everywhere else, use std::string.
- **Strongly typed enums**, for more type safety, and as an easy way to restrict the range of accepted values for an integer type; i.e., "int" has ~4 billion possible values, while a strongly typed enum could have only 4 or 5 values which are the only relevant ones. They are especially nice as keys for mapping to certain things, like file names or specific images, instead of using whole strings for mapping. It guarantees access correctness by restricting range on something that doesn't need to have 4 billion possible values!! It's an added bonus if a strongly typed enum does not implicitly convert to an integer type. I'd be even happier with the C++ enum classes if there was a simple way like in C# to get a set of all the values to iterate over, but I think that would require runtime type information.
- **container.at(index), instead of container[index]**, because bounds-checking is especially handy when working with the annoying zero-based indexing of C-like languages, where it's entirely unnecessary and bewildering to ask for the zeroth object in an array when every English speaking person (and others) asks for the first object and gets the first object! The only reason C is designed with zero-based indexing is to retain orthogonality with its base address + size multiplication for getting pointer offsets of various types without needing to do subtraction when wanting the first object.
- **nullptr**, for more type safety than NULL == ((void*)0). The std::nullptr_t is its own type which can't be confused with an integer type, but can be used anywhere a pointer is used.
- **std::array<type, count>**, because std::vector<type> uses dynamic allocation which is often times unnecessary, especially in situations where the vector's size is never intended to change after its first construction. This is another way of restricting ranges that should never be reached in the lifetime of an object, for instance when using bounds-checking. Unfortunately, I don't think std::arrays have any special constructors, like with iterators for example, so they have to be initialized manually with a loop for any data that's more than trivial. They can be copied to with std::copy, though, and that uses iterators.
- **std::uniform_distributions**, because rand() isn't professionally accurate enough. I haven't noticed any significant changes in performance from using a std::uniform_real_distribution paired with a std::default_random_engine in performance-critical applications.
- **Lambdas**, "[capture](args) -> return_type {}", for encapsulating expressions that are used in only one place, and to prevent temporary local variables from unnecessarily contaminating the function namespace. I don't particularly like the syntax outside of the curly braces though, because it seems to put the return type after the arguments, which is not found anywhere else in the language, from what I'm aware of. Why not design it like an inline function? My guess is that it's because writing the return type is optional, and if it were the first thing after the equals

sign, then it might cause some confusion if it were there sometimes and not other times, given its more "up front" placement in that case.

- **auto**, to reduce the number of times to re-type something, or to reduce the amount of screen/line space being used unnecessarily. Not super necessary on simple types like "int", or when Intellisense or other completion options are available. However, using "auto" on an integral type might reveal more specific information, like "std::vector<typename>::size_type" instead of an "int", which, is a really long type name, but... it's type safe and there's probably no loss of data, so... if "auto" is used in one place, it should be used elsewhere both for consistency, and to conceal the inherited long type names that are put together by combining calculations from various things. I don't think "auto" should be used for declaring function return values, but I suppose they get the idea for that from lambdas being able to have their return types deduced.
- **constexpr**, for guaranteed compile-time constant expressions (assuming it's for performance).
- **std::move()**, mostly for unique_ptrs, since they can't be copied, but also for moving things instead of copying them, which is a performance bonus.
- **static_assert**, for compile-time constant pre- and post-condition checking, which is nice for preventing bugs due to forgetting things. I actually use "assert()" more often, and I put it at the beginning and end of functions because I like runtime guarantees while a program is in development. Exception handling itself (try/catch) shouldn't be done too often in my opinion, unless something is REALLY exceptional.
- **override**, for being explicit about what virtual methods are being overridden. For some reason, VS2013's Intellisense doesn't reliably detect that a method is actually overriding a base virtual method, but it still compiles nonetheless.
- **Range-based for loop, i.e., "for (auto a : stuff)"**, instead of "for (auto it = ...begin(); ... ++it)". Less typing is good, and the auto can be replaced with something like a constant reference, just to be sure it's actually doing a constant reference and not a copy.
- **std::function**, for more flexibility with callbacks than function pointers. I think they should be okay with variadic arguments, like if they were going to be used generically for a "Button" type.
- **explicit**, for restricting constructors from doing implicit type conversions on their parameters, even if they're somewhat safe, just to keep everything running as intended. It's not really necessary to use this keyword unless the syntax is written with a lot of shortcuts.
- **std::list and std::forward_list**, for some handy extra container types. I think std::list is like a doubly linked list; for quick insertions but not-so-quick iteration.
- **SIMD intrinsic vector types like float4** (not until C++17 I think). These are good for when doing performance code, as they can more easily be recognized by the compiler as a vector operation, and can generally be converted to vector machine code. They are also good as a replacement for things like an RGBA color struct, where an int4 would just carry less weight with it, and would (hopefully) come with typical operators like addition and multiplication which would otherwise need to be explicitly written when using a struct or class. There would need to be some rules on how they are passed around though, because at a glance, they seem like something that would need to be passed by reference to keep from losing performance due to copy semantics. If I were to use them in C++, I would also want them in a header file instead of as a built-in type, because I think I would use them only for computer graphics applications, or applications where 3D math was the primary objective.

Things possibly worthy of more note:

- **Compile-time information**, like "std::is_constructable", etc.. The Visual Studio Intellisense shows that they are structs, but I don't know much more about them. I say that they are "compile-time" because they use templates.
- **Templates**. I don't find myself using these very much, and I'm not sure if I'm really missing out that much. When I want generic programming I like polymorphism and virtual methods. I guess templates just help out on the performance side of things, and for when an inheritance tree of objects isn't necessary. Lots of compile-time power from what I hear, and they also provide an entire Turing-complete meta-programming language.
- **Members from the "std::container_thing<type>::..." namespace**, like iterator, pointer, and const_reference, just to have the extra C++ wrapper around objects for type safety and flexibility. All of this compiles to relatively fast code anyway, so these features should be used for enhancing manageability! I believe in some cases, "auto" can deduce these long type names, too.
- **Initializer lists for constructors**. I think these are a reasonable idea, but partially unnecessary for the skilled programmer. They essentially allow the programmer to override the default constructor behavior for all non-intrinsic member variables for a class or struct. But by their existence, the programmer has two options for doing the exact same thing, so they will be left having to choose where their members get initialized; only in the initializer list, partially in the initializer list, or only in the constructor body (which also has the overhead of also calling default constructors for each member object, which I find unnecessary). Personally, I would rather have no initializer lists, and have it be a compile-time error if a non-intrinsic member is left uninitialized in the constructor body. They are initialized either way as it is already, so by making the programmer do it themselves is fine with me. I don't like the idea of constructors secretly calling other constructors and having things secretly get initialized twice.
- **std::chrono**, for timing operations accurately in a cross-platform way. No more Linux "sys/time.h" or Windows "QuerySomeFrequency…" functions necessary.

Things which should generally be avoided, but are not forbidden:

- **type variable_name = new type()**. With the addition of pointer wrappers like unique_ptr and shared_ptr, using manual memory allocation isn't in the best C++ style anymore, though it once was, as a type-safe leap forward from malloc().
- **C-style casts, like (void*)**. Sometimes the burden of "const_cast<void*>(reinterpret_cast<const void*>(thing))" just gets annoying at times, but it should still be used in cases where the amount to type is not unnecessarily extraordinary. Better to figure it out at compile time than runtime, and static_cast provides that very thing.
- **dynamic_cast**. This type of cast is a bit unsafe, mostly because it relies on runtime information, and can return null if unsuccessful. It isn't necessarily bad practice to use it, but it is bad practice to use it where it is not necessary. Overall, there are ways around requiring runtime type information; perhaps a change in design could prevent this cast from being needed. One workaround that comes to mind is to have derived classes implement an abstract method that returns the relevant information about their identity, like an enum. Another workaround is to have a factory that returns the requested instance of a class, and another is to use the "typeid()" operator to compare types of objects.

Things which should NOT be used unless for critical applications, like super-high speed performance (i.e., real-time ray tracing):

- **type variable_name[NUM_ELEMENTS]**. There is nothing valuable about prolonging the C style array in C++ now that there are convenience types like std::array that are designed to provide everything from before, with extra options like size querying and bounds-checking, all at the programmer's discretion. The programmer can use container[index] instead of container.at(index) if they want slightly more performance, but again, super-high speed performance isn't necessary for most things, so the bare-metal style of C isn't really something a modern C++ programmer should be going to now. Besides, is the behavior of arrays decaying to pointers and having to keep their size separate a desired trait of a language?
- **Anything weakly typed**. That includes, but is not limited to: 1) malloc, 2) C enums (not enum classes), and 3) #defined expressions, like "max" and "min". C is statically typed, but it is also weakly typed, which allows the programmer to basically break the type system wherever they go, and do things like move individual bytes around, which is considered perfectly acceptable at compile-time, because "the programmer knows what they're doing", even when they don't. There are C++ templates which implement what some #defined expressions can do, but with explicit specialization and compile-time type safety.
- **The C subset of C++**. C in itself is a form of premature optimization these days. We're not programming for 16-bit DOS computers with 32 megabytes of RAM anymore, and if we do need super-high speed performance code, we design the algorithm to run in parallel and we give it to the graphics card. With the release of compute languages like OpenCL, the only problem modern programmers really face now performance-wise is how to solve computationally-expensive problems that cannot be made parallel and require low latency, in which case the two best solutions are usually either reducing algorithmic complexity or making fine tuned, sometimes unreadable, changes to the code. The third solution is to improve hardware, but… that doesn't take a computer scientist to solve. Usually.

Side notes:

- Type systems are astronomically important! Live by them!
- Use the tools of C++ to discover bugs at compile time and to make busy-work things become automatic and type safe.
- All programmers can make mistakes if they're not perfectly careful! Take precautions by thinking ahead with using maintainable features that don't require babysitting and can simply be written and forgotten. One of the primary reasons C++ exists is to allow convenience without sacrificing performance, so these neat features working in the background should generally not add up to that much overhead.
- Overall, it seems like C++ is becoming more and more like Ada, especially with the C++17 suggested safety features (like pre- and post- conditions). Now all they need to do is add strict name equivalence, and I'll be happy.
- All these features I've listed are what C++ should have had ten or twenty years ago! C++ was too much like C for too long, and only now is it really starting to act like its own language. I say that with a boulder of salt. Maybe C++ has always been different enough from C, and it's been the programmers who treat C++ like C, or "C with Classes".