Functional programming in C++

**April 30, 2012 | By John Carmack**

<http://www.gamasutra.com/view/news/169296/Indepth_Functional_programming_in_C.php>

*[In this reprinted* [*#altdevblogaday*](http://altdevblogaday.com/) *in-depth piece, id Software co-founder and technical director John Carmack* [*looks at the value in programming in a functional style*](http://www.altdevblogaday.com/2012/04/26/functional-programming-in-c/) *with C++.]*  
  
Probably everyone reading this has heard "functional programming" put forth as something that is supposed to bring benefits to software development, or even heard it touted as a silver bullet. However, a trip to [Wikipedia](http://en.wikipedia.org/wiki/Functional_programming) for some more information can be initially off-putting, with early references to [lambda calculus](http://en.wikipedia.org/wiki/Lambda_calculus) and [formal systems](http://en.wikipedia.org/wiki/Formal_system). It isn't immediately clear what that has to do with writing better software.  
  
My pragmatic summary: A large fraction of the flaws in software development are due to programmers not fully understanding all the possible states their code may execute in. In a multithreaded environment, the lack of understanding and the resulting problems are greatly amplified, almost to the point of panic if you are paying attention. Programming in a functional style makes the state presented to your code explicit, which makes it much easier to reason about, and, in a completely pure system, makes thread race conditions impossible.  
  
I do believe that there is real value in pursuing functional programming, but it would be irresponsible to exhort everyone to abandon their C++ compilers and start coding in [Lisp](http://en.wikipedia.org/wiki/Lisp_%28programming_language%29), [Haskell](http://en.wikipedia.org/wiki/Haskell_%28programming_language%29), or, to be blunt, any other fringe language.  
  
To the eternal chagrin of language designers, there are plenty of externalities that can overwhelm the benefits of a language, and game development has more than most fields. We have cross-platform issues, proprietary tool chains, certification gates, licensed technologies, and stringent performance requirements on top of the issues with legacy codebases and workforce availability that everyone faces.  
  
If you are in circumstances where you can undertake significant development work in a non-mainstream language, I'll cheer you on, but be prepared to take some hits in the name of progress.  
  
For everyone else: ***No matter what language you work in,******programming in a functional style provides benefits. You should do it whenever it is convenient, and you should think hard about the decision when it isn't convenient****.* You can learn about lambdas, monads, currying, composing lazily evaluated functions on infinite sets, and all the other aspects of explicitly functionally oriented languages later if you choose.  
  
C++ doesn't encourage functional programming, but it doesn't prevent you from doing it, and you retain the power to drop down and apply SIMD intrinsics to hand laid out data backed by memory mapped files, or whatever other nitty-gritty goodness you find the need for.  
  
**Pure Functions**  
  
A pure function only looks at the parameters passed in to it, and all it does is return one or more computed values based on the parameters. It has no logical *side effects.* This is an abstraction of course; every function has side effects at the CPU level, and most at the heap level, but the abstraction is still valuable.  
  
It doesn't look at or update global state. it doesn't maintain internal state. It doesn't perform any IO. it doesn't mutate any of the input parameters. Ideally, it isn't passed any extraneous data – getting an allMyGlobals pointer passed in defeats much of the purpose.  
  
Pure functions have a lot of nice properties.  
  
Thread safety. A pure function with value parameters is completely thread safe. With reference or pointer parameters, even if they are const, you do need to be aware of the danger that another thread doing non-pure operations might mutate or free the data, but it is still one of the most powerful tools for writing safe multithreaded code.  
  
You can trivially switch them out for [parallel implementations](http://altdevblogaday.com/2011/11/22/parallel-implementations), or run multiple implementations to compare the results. This makes it much safer to experiment and evolve.  
  
Reusability. It is much easier to transplant a pure function to a new environment. You still need to deal with type definitions and any called pure functions, but there is no snowball effect. How many times have you known there was some code that does what you need in another system, but extricating it from all of its environmental assumptions was more work than just writing it over?  
  
Testability. A pure function has *referential transparency*, which means that it will always give the same result for a set of parameters no matter when it is called, which makes it much easier to exercise than something interwoven with other systems. I have never been very responsible about writing test code; a lot of code interacts with enough systems that it can require elaborate harnesses to exercise, and I could often convince myself (probably incorrectly) that it wasn't worth the effort.  
  
Pure functions are trivial to test; the tests look like something right out of a textbook, where you build some inputs and look at the output. Whenever I come across a finicky looking bit of code now, I split it out into a separate pure function and write tests for it. Frighteningly, I often find something wrong in these cases, which means I'm probably not casting a wide enough net.  
  
Understandability and maintainability. The bounding of both input and output makes pure functions easier to re-learn when needed, and there are less places for undocumented requirements regarding external state to hide.  
  
Formal systems and automated reasoning about software will be increasingly important in the future. [Static code analysis](http://altdevblogaday.com/2011/12/24/static-code-analysis/) is important today, and transforming your code into a more functional style aids analysis tools, or at least lets the faster local tools cover the same ground as the slower and more expensive global tools.  
  
We are a "Get 'er done" sort of industry, and I do not see formal proofs of whole program "correctness" becoming a relevant goal, but being able to prove that certain classes of flaws are not present in certain parts of a codebase will still be very valuable. We could use some more science and math in our process.  
  
Someone taking an introductory programming class might be scratching their head and thinking "aren't all programs supposed to be written like this?" The reality is that far more programs are [Big Balls of Mud](http://en.wikipedia.org/wiki/Big_ball_of_mud) than not. Traditional imperative programming languages give you escape hatches, and they get used all the time. If you are just writing throwaway code, do whatever is most convenient, which often involves global state.  
  
If you are writing code that may still be in use a year later, balance the convenience factor against the difficulties you will inevitably suffer later. Most developers are not very good at predicting the future time integrated suffering their changes will result in.  
  
**Purity In Practice**  
  
Not everything can be pure; unless the program is only operating on its own source code, at some point you need to interact with the outside world. It can be fun in a puzzly sort of way to try to push purity to great lengths, but the pragmatic break point acknowledges that side effects are necessary at some point, and manages them effectively.  
  
It doesn't even have to be all-or-nothing in a particular function. There is a continuum of value in how pure a function is, and the value step from almost-pure to completely-pure is smaller than that from spaghetti-state to mostly-pure. Moving a function towards purity improves the code, even if it doesn't reach full purity. A function that bumps a global counter or checks a global debug flag is not pure, but if that is its only detraction, it is still going to reap most of the benefits.  
  
Avoiding the worst in a broader context is generally more important than achieving perfection in limited cases. If you consider the most toxic functions or systems you have had to deal with, the ones that you know have to be handled with tongs and a face shield, it is an almost sure bet that they have a complex web of state and assumptions that their behavior relies on, and it isn't confined to their parameters. Imposing some discipline in these areas, or at least fighting to prevent more code from turning into similar messes, is going to have more impact than tightening up some low-level math functions.  
  
The process of refactoring towards purity generally involves disentangling computation from the environment it operates in, which almost invariably means more parameter passing. This seems a bit curious – greater verbosity in programming languages is broadly reviled, and functional programming is often associated with code size reduction.  
  
The factors that allow programs in functional languages to sometimes be more concise than imperative implementations are pretty much orthogonal to the use of pure functions — garbage collection, powerful built-in types, pattern matching, list comprehensions, function composition, various bits of syntactic sugar, etc. For the most part, these size reducers don't have much to do with being functional, and can also be found in some imperative languages.  
  
You *should* be getting irritated if you have to pass a dozen parameters into a function; you may be able to refactor the code in a manner that reduces the parameter complexity.  
  
The lack of any language support in C++ for maintaining purity is not ideal. If someone modifies a widely used foundation function to be non-pure in some evil way, everything that uses the function also loses its purity. This sounds disastrous from a formal systems point of view, but again, it isn't an all-or-nothing proposition where you fall from grace with the first sin. Large scale software development is unfortunately statistical.  
  
It seems like there is a sound case for a pure keyword in future C/C++ standards. There are close parallels with const – an optional qualifier that allows compile time checking of programmer intention and will never hurt, and could often help, code generation. The D programming language [does offer a pure keyword](http://www.d-programming-language.org/function.html). Note their distinction between weak and strong purity – you need to also have const input references and pointers to be strongly pure.  
  
In some ways, a language keyword is over-restrictive — a function can still be pure even if it calls impure functions, as long as the side effects don't escape the outer function. Entire programs can be considered pure functional units if they only deal with command line parameters instead of random file system state.  
  
**Object Oriented Programming**  
  
[***Michael Feathers*** *@****mfeathers***](https://twitter.com/#%21/mfeathers)*OO makes code understandable by encapsulating moving parts. FP makes code understandable by minimizing moving parts.*  
  
The "moving parts" are mutating states. Telling an object to change itself is lesson one in a basic object-oriented programming book, and it is deeply ingrained in most programmers, but it is anti-functional behavior. Clearly there is some value in the basic OOP idea of grouping functions with the data structures they operate on, but if you want to reap the benefits of functional programming in parts of your code, you have to back away from some object-oriented behaviors in those areas.  
  
Class methods that can't be const are not pure by definition, because they mutate some or all of the potentially large set of state in the object. They are not thread safe, and the ability to incrementally poke and prod objects into unexpected states is indeed a significant source of bugs.  
  
Const object methods can still be technically pure if you don't count the implicit *const this* pointer against them, but many object are large enough to constitute a sort of global state all their own, blunting some of the clarity benefits of pure functions. Constructors can be pure functions, and generally should strive to be – they take arguments and return an object.  
  
At the tactical programming level, you can often work with objects in a more functional manner, but it may require changing the interfaces a bit. At id we went over a decade with an idVec3 class that had a self-mutating void Normalize() method, but no corresponding idVec3 Normalized() const method. Many string methods were similarly defined as working on themselves, rather than returning a new copy with the operation performed on it – ToLowerCase(), StripFileExtension(), etc.  
  
**Performance Implications**  
  
In almost all cases, directly mutating blocks of memory is the speed-of-light optimal case, and avoiding this is spending some performance. Most of the time this is of only theoretical interest; we trade performance for productivity all the time.  
  
Programming with pure functions will involve more copying of data, and in some cases this clearly makes it the incorrect implementation strategy due to performance considerations. As an extreme example, you can write a pure DrawTriangle() function that takes a framebuffer as a parameter and returns a completely new framebuffer with the triangle drawn into it as a result. Don't do that.  
  
Returning everything by value is the natural functional programming style, but relying on compilers to always perform [return value optimization](http://en.wikipedia.org/wiki/Return_value_optimization) can be hazardous to performance, so passing reference parameter for output of complex data structures is often justifiable, but it has the unfortunate effect of preventing you from declaring the returned value as const to enforce [single assignment](http://en.wikipedia.org/wiki/Single_assignment#Single_assignment).  
  
There will be a strong urge in many cases to just update a value in a complex structure passed in rather than making a copy of it and returning the modified version, but doing so throws away the thread safety guarantee and should not be done lightly. List generation is often a case where it is justified. The pure functional way to append something to a list is to return a completely new copy of the list with the new element at the end, leaving the original list unchanged. Actual functional languages are implemented in ways that make this not as disastrous as it sounds, but if you do this with typical C++ containers you will die.  
  
A significant mitigating factor is that performance today means parallel programming, which usually requires more copying and combining than in a single threaded environment even in the optimal performance case, so the penalty is smaller, while the complexity reduction and correctness benefits are correspondingly larger.  
  
When you start thinking about running, say, all the characters in a game world in parallel, it starts sinking in that the object-oriented approach of updating objects has some deep difficulties in parallel environments. Maybe if all of the object just referenced a read only version of the world state, and we copied over the updated version at the end of the frame… Hey, wait a minute…  
  
**Action Items**  
  
Survey some non-trivial functions in your codebase and track down every bit of external state they can reach, and all possible modifications they can make. This makes great documentation to stick in a comment block, even if you don't do anything with it. If the function can trigger, say, a screen update through your render system, you can just throw your hands up in the air and declare the set of all effects beyond human understanding.  
  
The next task you undertake, try from the beginning to think about it in terms of the real computation that is going on. Gather up your input, pass it to a pure function, then take the results and do something with it.  
  
As you are debugging code, make yourself more aware of the part mutating state and hidden parameters play in obscuring what is going on.  
  
Modify some of your utility object code to return new copies instead of self-mutating, and try throwing const in front of practically every non-iterator variable you use.  
  
Additional references:  
  
<http://www.haskell.org/haskellwiki/Introduction>  
  
<http://lisperati.com/>  
  
<http://www.johndcook.com/blog/tag/functional-programming/>  
  
<http://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf>  
  
<http://channel9.msdn.com/Shows/Going+Deep/Lecture-Series-Erik-Meijer-Functional-Programming-Fundamentals-Chapter-1>  
  
<http://www.cs.utah.edu/~hal/docs/daume02yaht.pdf>  
  
<http://www.cs.cmu.edu/~crary/819-f09/Backus78.pdf>  
  
<http://fpcomplete.com/the-downfall-of-imperative-programming/>  
  
*[This piece was reprinted from* [*#AltDevBlogADay*](http://altdevblogaday.com/)*, a shared blog initiative started by* [*@mike\_acton*](http://www.twitter.com/mike_acton) *devoted to giving game developers of all disciplines a place to motivate each other to write regularly about their personal game development passions.]*