Subject: Re: Should the pick semantics be changed?
Posted by piotr5 on Tue, 25 Mar 2014 18:35:14 GMT
View Forum Message <> Reply to Message

you are almost correct on what the actual problem is, but your solution shows you think in terms of low-level languages. the goal of c++ is to abstract away the optimizations so that they can be handled seperately instead of polluting your code with it. in case of returning objects from a function this "promise" gets broken. u++ solved this problem by saying that all initializations of new objects happen by moving the contents of an object and thereby destroying the originating object. i.e. all containers in u++ are treated (on low-level) as if they were just references, while on high-level type-safety makes sure you don't use old objects that have been altered in the newly created container. now std-c++11 introduced something similar to what u++ has, but on the level of the compiler.

in your example, you would need to encapsulate the class C into another class which holds a const reference to it. then in getC you create new objects pointing at the C object you choose. this way you avoid copying C. additionally you have less need for whatever garbage collection since the old C object doesn't move around in memory, so you have no new fragmentation except for some gaps of pointer-size which will find some use quickly.

however, this thread talks about objects that are not constant. so you have this collection of static predefined objects, and you don't just return some constant stuff, you give ownership and control over C to a particular encapsulating object. this collection of static objects you had, now contains one object less, you already gave away exclusive control to someone for this removed object. inside memory nothing changed, the object you returned still occupies the same memory it always did, except for some overhead, it just isn't available anymore. for example if the compiler would generally place static objects inside of the same memory as the function's code (which is nonsense since code is usually read-only for the processor) then the returned object would work on that area instead on heap or stack or whatever.

you are wrong when you think my proposal would be about return-types, it's rather about object initialization. there is an inconsistency in g++: when you write "new C(getC());" then on the heap the C object is initialized with r-value, it's syntactic sugar for "new C(move(getC));". however, if inside getC() you write "C o; createCBasedOnType(type,o); return o;" then this isn't syntactic sugar for "C o; createCBasedOnType(type,o); return move(o);". in the first case the syntactic sugar is rationalized by the impermanence of the C object getC() sends to the initializer. but also the object o which is the return-value of getC() is impermanent, it gets destroyed immediately after the return-value has been created. it is an inconsistency that both cases aren't treated the same in terms of syntactic sugar! it's not really a bug, but it's counterintuitive.