

Re: why still use C?

Source: http://coding.derkeiler.com/Archive/C_CPP/comp.lang.c/2003-10/0857.html

From: Glen Herrmannsfeldt (gah_at_ugcs.caltech.edu)

Date: 10/06/03

Date: Mon, 06 Oct 2003 05:28:35 GMT

"Sidney Cadot" <sidney@jigsaw.nl> wrote in message
news:blqeq9\$1lc\$1@news.tudelft.nl...

> *Hi cody,*

>

>> *no this is no trollposting and please don't get it wrong but iam very*

>> *curious why people still use C instead of other languages especially
C++.*

>

> *I can't answer for people in general of course, but as a moderately able*

> *C programmer with a thorough dislike of C++ I can try to explain what my*

> *motives are.*

>

>> *i heard people say C++ is slower than C but i can't believe that. in*

pieces

>> *of the application where speed really matters you can still use "normal"*

>> *functions or even static methods which is basically the same.*

>

> *A few years ago I did some timing and (counter to my intuition) I found*

> *that, indeed, it didn't make a difference, as you point out. One*

> *remarkable thing was that the C++ executables for my (small) benchmarks*

> *were quite a bit larger, which may be relevant for embedded applications.*

OOP in general tends to be slower. The process of allocating and
deallocating memory, including finding a good sized region to allocate,
takes time. As you say, though, one can do non-OOP in C++, and, with some
work, OOP in C.

>> *in C there arent the simplest things present like constants, each struct
and*

>> *enum have to be prefixed with "struct" and "enum". iam sure there is
much*

>> *more.*

As if the compiler didn't know... In PL/I structures can be referenced in
any unambiguous way. I don't know if that leads to more bugs, or makes
programs more or less readable, though. It does seem strange that you have
to keep reminding the compiler that something is a struct.

comp.lang.c: Re: why still use C?

- > *These are some areas where I would agree that yes, C++ is (a bit)*
- > *cleaner than C. Another example is declaring variables inside for()*
- > *statements and such, this can truly help readability, and limiting the*
- > *scope of a local variable if possible is a good thing. Note that many of*
- > *these (almost cosmetic) changes have made their way back into C99.*
- >
- > > *i don't get it why people program in C and faking OOP features(function*
- > > *pointers in structs..) instead of using C++. are they simply masochists*
- or
- > > *is there a logical reason?*

Note, though, that Java is much closer to C than C++ is, despite the similarity of names. If you like C, but want an OO language, Java should be your choice.

- > *In all honesty I think that many people who prefer C over C++ don't*
- > *quite get what all the fuzz is about in OOP (I know I don't). In*
- > *principle there are sound advantages to grouping together structs and*
- > *their associated method functions from a design perspective. Inheritance*
- > *and polymorphism have an important part to play as well, especially in*
- > *some areas of application (such as GUIs).*

For some kinds of programming projects, yes.

(snip)

- >. *However, as a programmer I am no*
- > *longer completely in the driver's seat as I am with C. Looking at a C*
- > *program, I can have a straightforward and relatively accurate mental*
- > *picture of what the actual machine code produced by the compiler will*
- > *look like. With OOP and C++, that's no longer true, especially with code*
- > *that uses all the available C++ features including exceptions and*
- templates.

Well, some people consider C as a glorified assembler. It isn't quite that, especially as it has changed over the years, but not so far off.

- > *One generic complaint I have with OOP (not limited to C++) is that I can*
- > *no longer look at a code fragment and reconstruct the execution flow in*
- > *my head, because of polymorphism and operator overloading; in C (unless*
- > *you're doing funky stuff with threads or longjumps), the execution flow*
- > *is pretty much known at compile-time, and can be reproduced from the*
- > *code. I happen to like that.*

Well, with library functions in general you don't know what is inside the function. If you are writing the whole program yourself then the abstraction is less useful. If different people are working on different parts then abstraction means you need to know less about the specific features of those parts. The interface is narrower, which sometimes decreases efficiency. (It may take more calls to get something done, or more things done than are really needed.)

Re: why still use C?

- > *Of course OOP proponents will counter that this is in fact the entire*
- > *point of OOP: one should no longer be thinking in terms of structural*
- > *execution flow, but rather in terms of objects with a well-defined*
- > *behaviorial 'contract', that can be triggered by invoking methods.*

(snip)

- > *As for exceptions, you may know Dijkstra's paper "Goto's considered*
- > *harmful". In this paper he has a number of points that I would subscribe*
- > *to, concerning the ability of the human programmer to read the meaning*
- > *of a piece of code from the source. In essence, he argues that GOTO*
- > *statements destroy this possibility.*
- >
- > *I would argue that exceptions are "goto's on steroids". Since exceptions*
- > *are allowed to cross function-call boundaries, execution flow becomes*
- > *very non-transparent – at least to me! This is a similar objection I*
- > *have with polymorphism as described above.*

Well, there is that. But the name, exception, gives you some idea of their use. They should be used for exceptional things. In compilers sometimes there is nothing that can be done. Especially in recursive descent compilers it may be that the only thing to do is declare an error and go onto the next statement. That requires crossing function call boundaries, but it is easy to understand what is happening. The C setjmp/longjmp has a similar use, and is similarly non-transparent.

(snip)

- > *To summarize I would say C++ with its feature set is just too*
- > *complicated, as a language design I feel it has failed. One has to keep*
- > *in mind that a programming language is a tool to make programs. If a*
- > *tool has a significant chance of being unintentionally misused (with*
- > *possibly disastrous results), it's not a good tool. I will stick with*
- > *something I actually (more or less) understand, which is C.*

Well, much of the idea of C is to be simple. I learned PL/I as my first structures language, and I still prefer it, in some ways, to C. PL/I is complicated, almost by design. (It was designed to include features from Algol, Fortran, and COBOL, all in one language.) C string handling is simple in design, somewhat efficient, but so easy to do wrong. Again, PL/I was designed to be complicated, but such that you didn't need to learn parts you didn't need to use. That required no reserved words. (If you didn't know about a feature how could you know not to use the words?) Writing simple programs is pretty simple. The dynamic memory features of C are fine once you are used to them, but pretty strange until then.

— glen