Is C++ suitable for embedded systems?

A common question, here and elsewhere. Is C++ suitable for embedded systems?

Microcontrollers? RTOSes? Toasters? Embedded PCs?

Is OOP useful on microcontrollers?

Does C++ remove the programmer too far from the hardware to be efficient?

Should Arduino's C++ (with no dynamic memory management, templates, exceptions) be considered as "real C++"?

(Hopefully, this wiki will serve as a place to contain this potential holy war)

c++ embedded

edited Sep 17 '12 at 9:01

community wiki
5 revs, 3 users 69%
Toby Jaffey

4 Quick question: when you say *embedded*, do you mean microcontroller? microprocessor? embedded x86 / embedded PC? – J. Polfer Jun 15 '10 at 16:07

1 I didn't intend to cause a holy war; the intent was to learn what your arguments were against it. – J. Polfer Jun 15 '10 at 16:08

2 It's come up in several questions before, so I thought a central place would be good. – Toby Jaffey Jun 15 '10 at 16:13

4 C++ vs embedded is a contentious topic. I have a strong opinion, but I didn't think it was fair to raise a question and play at scoring points. I hope that a community wiki will make for a more balanced discussion. – Toby Jaffey Jun 15 '10 at 16:36

10 This is a bad question since "embedded" is a meaningless attribute in deciding whether a particular language and its associated baggage is suitable. The point is small versus large systems, where small systems aren't running a OS, have limited memory, may not be von-Neuman, may have various hardware restrictions on call stacks, data stacks, you can't just dynamically allocate a Mb or even a kb, etc. Most microcontrollers are "small" systems. Single board computers are usually embedded, but are generally "large" systems. – Olin Lathrop Jun 28 '12 at 11:42

16 Answers

Yes, C++ is still useful in embedded systems. As everyone else has said, it still depends on the system itself, like an 8-bit uC would probably be a no-no in my book even though there is a compiler out there and some people do it (shudder). There's still an advantage to using C++ even when you scale it down to something like "C+" even in a 8-bit micro world. What do I mean by "C+"? I mean don't use new/delete, avoid exceptions, avoid virtual classes with inheritance, possibly avoid inheritance all together, be very careful with templates, use inline functions instead of macros, and use `const` variables instead of `#defines` .

I've been working both in C and C++ in embedded systems for well over a decade now, and some of my youthful enthusiasm for C++ has definitely worn off due to some real world problems that shake one's naivete. I have seen the worst of C++ in an embedded systems which I would like to refer to as "CS programmers gone wild in an EE world." In fact, that is something I'm working on with my client to improve this one codebase they have among others.

The danger of C++ is because it's a very very powerful tool much like a two-edged sword that can cut both your arm and leg off if not educated and disciplined properly in it's language and general programming itself. C is more like a single-edged sword, but still just as sharp. With C++ it's too easy to get very high-levels of abstraction and create obfuscated interfaces that become meaningless in the long-term, and that's partly due to C++ flexibility in solving the same problem with many different language features(templates, OOP, procedural, RTTI, OOP+templates, overloading, inlining).

I finished a two 4-hour seminars on Embedded Software in C++ by the C++ guru, Scott Meyers. He pointed out some things about templates that I never considered before and how much more they can help creating safety-critical code. The jist of it is, you can't have dead code in software that has to meet stringent safety-critical code requirements. Templates can help you accomplish this, since the compiler only creates the code it needs when instantiating templates. However, one must become more thoroughly educated in their use to design correctly for this feature which is harder to accomplish in C because linkers don't always optimize dead code. He also demonstrated a feature of templates that could only be accomplished in C++ and would have kept the Mars Climate Observer from crashing had NASA implemented a similar system to protect units of measurement in the calculations.

Scott Meyers is a very big proponent on templates and judicious use of inlining, and I must say I'm still skeptical on being gung ho about templates. I tend to shy away from them, even though he says they should only be applied where they become the best tool. He also makes the point that C++ gives you the tools to make really good interfaces that are easy to use right and make it hard to use wrong. Again, that's the hard part. One must come to a level of mastery in C++ before you can know how to apply these features in most efficient way to be the best design solution.

The same goes for OOP. In the embedded world, you must familiarize yourself with what kind of code the compiler is going to spit out to know if you can handle the run-time costs of run-time polymorphism. You need to be willing to make measurements as well to prove your design is going to meet your deadline requirements. Is that new InterruptManager class going to make my interrupt latency too long? There are other forms of polymorphism that may fit your problem better such as link-time polymorphism which C can do as well, but C++ can do through the Pimpl design pattern (Opaque pointer).

I say that all to say, that C++ has its place in the embedded world. You can hate it all you want, but it's not going away. It can be written in a very efficient manner, but it's harder to learn how to do it correctly than with C. It can sometimes work better than C at solving a problem and sometimes expressing an better interface, but again, you've got to educate yourself and not be afraid to learn how.

edited May 2 '13 at 18:59

community wiki
6 revs, 3 users 86%
Jay Atkinson

This goes inline with what I have read from other embedded systems consultants. I have always been taught that with C, you cut your self in little ways constantly, but a bug in C++ is going to be more rare, but when you screw up you are going to lose a leg. Thank you for writing a clear response with some expertise that I do not have. – Kortuk ♦ Jun 16 '10 at 14:23

I had been informed that you really need to spend time and get trained in proper techniques before charging headfirst into C++. – Kortuk ♦ Jun 16 '10 at 14:23

2 On the note of Computer Science majors going crazy in EE land. At my job the worst piece of code we have was written by a CS major. We spent forever trying to teach him what the hardware was. He made a structured system using UML and built the entire system based on it in java using proper inheritance and so forth. It worked, until anything changed, and then it was a bad patch job to add features, or a complete redesign. The code is almost not usable because of how thoroughly he obfuscated the whole thing with inheritance. – Kortuk ♦ Jun 16 '10 at 14:36

2 It's not just bugs that bite you, but unmaintainable code that will bite you too. Sure, when you start that project out using those neat C++ features everything goes swimmingly, but after 2 or 3 years entropy kicks in if no serious effort is put into refactoring as you develop. That's what I'm facing right now..the code rots faster over time in C++. – Jay Atkinson Jun 16 '10 at 17:22

UML and statemachines. You really need to look into Miro Samek's stuff at state-machine.com. He's built an efficient system that's easy to refactor and change, but it does take some time to grok it. – Jay Atkinson Jun 16 '10 at 17:24

C++ is absolutely suitable for embedded systems. I now use the presence/absence of good development tools (or lack thereof) as my primary criterion for whether or not to use a particular microprocessor.

Areas of C++ that are good to use on embedded systems because they have low resource costs:

- modularity brought by good use of classes/structures
- templates **if** the compiler does a good job of compiling them efficiently. Templates are a good tool for bringing reuse of algorithms to different data types.

OK areas:

- virtual functions -- I used to be against this, but the resource cost is very small (one vtable per *class*, not per object; one pointer to the vtable per object; one dereferencing operation per virtual function call) and the big advantage of this is that it allows you to have an array containing several different types of objects w/o having to know what type they are. I used this recently to have an array of objects each representing an I2C device, each with separate methods.

Areas not to use, mostly because of the run-time overhead that is unacceptable on small systems:

- dynamic memory allocation -- others have mentioned this, but another important reason *not* to use dynamic memory allocation is that it represents uncertainty in timing; many reasons to use embedded systems are for real-time applications.
- RTTI (run time type information) -- the memory cost is rather large
- exceptions -- a *definite* no-no, because of the execution speed hit

edited Jan 24 '15 at 22:41 — community wiki
2 revs, 2 users 91%
Jason S

Thanks for the input. Interesting and very similar to what I have read. – Kortuk ♦ Jun 20 '10 at 20:38

1 Actually dynamic memory allocation is fine, and sometimes unavoidable. It is dynamic memory DEallocation (and subsequent re-use) that is the problem. RTTI is memory hog, I agree on that one. But what is the problem with exceptions? – Wouter van Ooijen Sep 6 '11 at 13:18

1 @WoutervanOoijen: The problem with exceptions is that if `foo` calls `bar` within a `try` / `catch` block, and `bar` creates some objects and calls `boz`, which throws an exception, the system has to somehow call the destructors for the objects `bar` created before returning control to `foo`. Unless exceptions are completely disabled, `bar` will have no way of knowing whether `boz` might throw any, and must thus include extra code to allow for that possibility. I'd like to see a variation of C++ with "checked exceptions" to deal with that; if routines which could allow exceptions to escape... – supercat Jun 28 '12 at 16:04

1 ...had to be declared as such, then it would only be necessary for the compiler to include exception-handling code in the callers of such routines. To be sure, having to add in all the required declarations, while hopefully avoiding unnecessary ones, would be somewhat burdensome, but it would allow exceptions to be used in places where they are useful, without adding overhead where they are not. – supercat Jun 28 '12 at 16:07

3 @WoutervanOoijen: Incidentally, if I were designing the ABI for such exception handling on the ARM, I would specify that code which calls a routine that may exit via exception should have R14 point to an address two bytes before the desired return address (this would occur naturally if the caller followed the CALL instruction with a 16-bit word). The called routine would then exit normally via `add r15,r14,#2` instead of `mov r15,r14` ; to exit via exception, `ldrhs r0,[r14] / add r15,r14,r0` . Zero cycle cost for the normal exit, and no stack-frame restrictions. – supercat Jun 28 '12 at 16:14

Yes, C++ is certainly suitable for embedded systems. First let's clear up a couple of misconceptions about the difference between C and C++:

In an embedded micro, you're always going to need to use high level languages carefully if you're concerned about time or space constraints. For example, many MCUs don't handle pointers well, and so are very inefficient when using the stack. This means you have to be careful about passing variables to functions, using arrays and pointers, and recursion. A simple line of C like:

```
a[i] = b[j] * c[k];
```

can generate about 4 pages of instructions depending on the nature of those variables.

Whenever you're using *any* high level language, and you're concerned about time and space constraints, then you need to know how *every* feature of that language translates into machine instructions on your MCU (at least, every feature that you use). This is true for C, C++, Ada, whatever. Probably all languages will contain features that don't translate efficiently on small MCUs. Always check the disassembly listings to make sure the compiler's not generating reams of instructions for something trivial.

Is C suitable for embedded MCUs? Yes, as long as you keep an eye on the generated code.
Is C++ suitable for embedded MCUs? Yes, as long as you keep an eye on the generated code.

Here's why I think that C++ is better than C even on 8-bit MCUs: C++ provides improved support for:

- Data hiding
- Stronger typing / checking
- Multi-peripheral transparency using classes
- Templates (as always if used carefully)
- Initialisation lists
- const

None of these features are any heavier than typical features of C.

As you move up to 16 or 32 bit MCUs, then it starts to make sense to use heavier features of C (stack, heap, pointers, arrays, printf, etc.) In the same way, on a more powerful MCU is becomes appropriate to use heavier features of C++ (stack, heap, references, STL, new/delete).

So, there's no need to shudder at the thought of C++ on a PIC16. If you know your language and your MCU properly, then you will know how to use them both effectively together.

answered Feb 20 '12 at 22:51 community wiki Rocketmagnet

2 This is a very well-expressed and reasonable answer to the question. +1 Cheers! – vicatcu Feb 23 '12 at 15:20

" `a[i] = b[j] * c[k];` can generate about 4 pages of instructions depending on the nature of those variables." If your MCU/compiler does this, it is because you are using some garage hobbyist CPU from the 80s. – Lundin Jun 4 '15 at 6:27

@Lundin - Sigh. No, it means you're using a cheap little MCU which is designed to be as small and as cheap as possible, not to have complex things like stack indexing. – Rocketmagnet Jun 10 '15 at 17:42

1 @Rocketmagnet Ok maybe in the 1990s? Nowadays crappy 8 bitters come at the same price as a 32 bitters. Only reason left for picking the former is current consumption. And regarding those extra-crappy 8-bitters with no stack: if you writing C instead of assembler for such a limited MCU, you are likely doing it wrong. The 4 pages generated is then your own fault for writing too complex programs for the CPU, and essentially C is the wrong tool for the task. (I have done this in the past on Freescale RS08, it was a very stupid idea.) – Lundin Jun 11 '15 at 9:33

I always find these debates entertaining to read. Not so much for the intellectual discussion about the pros and cons of the various available languages but because you can usually peg someones stance on the topic based on their job/experience/area of interest. Its right up there with the "premature optimization" arguments were the CS majors and maintenance programmers quote Knuth left and right and those who work in the real world where performance matters think they're all crazy (i'm a member of the later group to be fair).

At the end of the day, you can develop excellent software in C or C++ or *insert language here*. It comes down to the capabilities of the developer not the language. Being an expert in a language is usually only required if you've chosen the wrong language to begin with and now need to warp it into solving your problem, in most cases these are the only situations where you need to dive into obscure features or compiler tricks to accomplish the goal.

I often hear people start these arguments as "i'm an expert in language X and blah blah" I honestly immediately discredit these people because, in my opinion, they've already approached the problem from the wrong angle and everything after that is tainted by their desire to use their tool to solve the problem and show how 'cool' it is.

I so often watch developers choose a tool set first and attempt to bend it to their problem second, which is completely wrong and results in crap solutions.

As I mentioned in a comment to another answer, these language wars often devolve into arguing that language X allows the programmer to do more dumb things. While entertaining to read, all these statements really mean is that you have a problem hiring good developers and need to address that issue directly rather than trying to band aid the situation by continuing to hire bad developers and choosing tools such that they can do as little damage as possible.

In my opinion good developers, be it software or hardware development, research the problem, architect a solution and find the tools that allow them to express the solution in the 'best way'. It shouldn't matter if the required tool is something you've never used before, after you've used 3-4 languages/development tools for projects picking up a new one should have a minimal impact on your development time.

Of course, 'best way' is a subjective term and also needs to be defined in the research phase. One needs to consider a multitude of issues: performance, ease of expression, code density, etc based on the problem at hand. I didn't include maintainability in that list for a reason, i don't care what language you choose, if you've chosen the proper tool and taken the time to understand the problem this should come 'for free'. Difficult to maintain code is often the result of choosing the wrong tool or a poor system structure, this results in an ugly hacky mess to make it work.

Claiming any language is 'better' than any other is silly without defining a particular problem of interest. An object oriented approach is not always better than a functional approach. There are some problems that lend themselves very well to an object oriented design paradigm. There are many that do not. The same statement can be made about many language features that people seem to enjoy harping on.

If your spending more than 20% of your time on a problem actually typing code, your probably producing a very poor system, or have very poor developers(or your still learning). You should be spending the majority of your time up front diagramming the problem and determining how various pieces of the application interact. Sticking a group of talented developers in a room with a marker board and a problem to solve and telling them they are not allowed to write any code or

choose any tools till they feel comfortable with the entire system will do more to improve the quality of the output and speed development than choosing any hot new tool guaranteed to improve development time. (look up scrum development as a reference for the polar opposite to my argument)

Often the unfortunate reality is that many businesses can only measure the value of a developer by the number of lines written, or by seeing 'tangible output'. They view the 3 weeks in a room with a marker board as a loss in productivity. Developers are often forced to speed through the 'thought' stage of development or are forced into using a tool set by some political issue within the company, "my boss's brother works for IBM so we can only use their tools", that kind of rubbish. Or worse, you get a constantly changing set of requirements from the company because they are not capable of doing proper market research or don't understand the impact of changes on the development cycle.

Sorry for being slightly off topic with this rant, I have quite strong opinions on this topic.

edited Jun 17 '10 at 2:40 — community wiki, 3 revs, Mark

+1, I enjoyed it, although a bit long. I only use C at work, but I am really trying to move to C++ because there are many bonuses. I have my biases, mostly because of what has gone wrong, or the fact that I have a boss who changes specs within 2 days of telling us what is needed. I am still learning, and I expect to feel that way for about another 4-5 years. I will still learn after that point, but my company does not have a style guide that is followed or an inspection process and I am trying to get both. – Kortuk ♦ Jun 17 '10 at 1:00

I think you hit the nail on the head with your comment on bending the tool around the problem. It reminds me of the saying, "When all you have is a hammer, everything looks like nail!" Was that a pun?;-) TDD is definitely flies in the face of up front thought, and I don't think I'll ever get comfortable with the idea of making tests to create the design...especially in embedded systems. How in the world can you mock a hardware piece to unit-test a device driver without wasting a lot of time on the mock piece?! – Jay Atkinson Jun 17 '10 at 1:08

1 Now, I'm not knocking unit tests on an application level (above driver) on certain embedded systems. There is some value of the instant feedback of unit-testing and rooting out bugs early in the development phase, but the whole TDD paradigm to give birth to the design seems to me a little braindead. I prefer to take some time to "think" about the problem and diagram it out either in my head, on paper, or on a whiteboard, before starting to code. I also think TDD encourages the market to not do upfront research on requirements, because it's supposed to help with constant requirement change. – Jay Atkinson Jun 17 '10 at 1:13

1 And to put a final note on my super long comment.. We don't need language experts to work the design. We need expert designers who can work the language(s). – Jay Atkinson Jun 17 '10 at 1:21

sorry for the epically long rant, i tend to get rather frantic when i see an issue debated heavily when i see it as a simple result of a deeper problem that i think should be the real topic of discussion. – Mark Jun 17 '10 at 1:52

In my experience, C++ is usually ill-suited to small embedded systems. By which I mean, microcontrollers and OS-less devices.

Many C++ OOP techniques rely on dynamic memory allocation. This is often missing in small systems.

STL and Boost really demonstrate the power of C++, both are huge in footprint.

C++ encourages the programmer to abstract away the machine, where in constrained systems it has to be embraced.

Last year, I ported a commercial remote desktop product to mobile phones. It was written in C++ and ran on Windows, Linux and OSX. But, it relied heavily on STL, dynamic memory and C++ exceptions. To get it going on WinCE, Symbian and OS-less environments a C rewrite was the sanest option.

edited Jun 21 '10 at 0:48 — community wiki, 5 revs, Toby Jaffey

I agree in reference to small systems, but I think we have different definitions of small systems. When you have 1kB of ROM and well written C code takes all but 1 byte of ROM, that is a small system. – Kortuk ♦ Jun 15 '10 at 16:20

5 I am not arguing that C cannot have a smaller footprint, but you could have used C++ still and obtained a very similar result for designing for what was just discussed. I think the issue is most OOP programmers are used to systems with dynamic memory and using very inefficient constructs, resulting in completely useless code for lower power systems. – Kortuk ♦ Jun 15 '10 at 16:27

2 so what your saying is you don't want to use C++, you want to use something between C and C++ (lets just call it C+?). In that case i agree, theres a lot of crap in C++ people use just because its available, not because its optimal. Almost any language is capable of producing good, fast code, its a matter of how its used. Most holy wars over languages are not a result of the languages capabilities but an argument over how easy it is for an idiot to do idiotic things, which is an idiotic argument really :p – Mark Jun 16 '10 at 4:00

1 "Most holy wars over languages are not a result of the languages capabilities but an argument over how easy it is for an idiot to do idiotic things, which is an idiotic argument really." Was a very enjoyable

sentence. I need your last name so I can quote that one. – Kortuk ♦ Jun 16 '10 at 14:27

1 I really do not use the dynamic memory in C either though. There is nowhere that I have to have it. In long term I have read that it can get very very segmented and start causing problems. I need to have very clearly designed cases for running out of memory, and I need to be able to monitor exactly how much is left. – Kortuk ♦ Jun 16 '10 at 14:34

Any language can be suitable for an embedded system. Embedded just means: part of a larger apparatus, as opposed to a free-to-use computer.

The question has more relevance when asked for a *(hard-)real-time* or *limited-resources* system.

For a real-time system C++ is one of the highest languages that is still appropriate when programming for stringent time constraints. With the exception of heap use (free operator) it has no constructs that have an indeterminate execution time, so you can test whether your programm fulfills its timing requirements, and with some more experience you might even predict it. Heap use should of course be avoided, although the new operator can still be used for one-time allocation. The constructs that C++ offers over C can be put to good use in an embedded system: OO, exceptions, templates.

For very resource-limited systems (8-bit chips, less than a few Kb of RAM, no acessible stack) full C++ might be ill-suited, although it might still be used as a 'better C'.

I think it unfortunate that Ada seems to be used only in some niches. In a lot of ways it is a Pascal++, but without the burden of being upwards compatible with a language that was already a serious mess to begin with. (edit: the serious mess is of course C. Pascal is a beautiful but somewhat impractical language.)

==========

EDIT: I was typing an answer to new question ("In which cases is C++ necessary when we are programming microcontrollers"?) that was closed referring to this one, so I'll add what I wrote:

There is never an all-overruling reason for the use of *any* programming language, but there can be arguments that have more or less weight in a particular situation. Discussions about this can be found in a lot of places, with positions taken that range from "never use C++ for a micro-controller" to "always use C++". I am more with the last position. I can give some arguments, but you'll have to decide for yourself how much weight they carry in a particular situation (and in which direction).

- C++ compilers are more rare than C compilers; for some targets (for instance 12 and 14 bit core PICs) there are no C++ compilers at all.
- (good) C++ programmers are more rare than (good) C programmers, especially among those that are also (somewhat) knowledgeable in electronics.
- C++ has more constructs than C that are not appropriate for small systems (like exceptions, RTTI, frequent use of the heap).
- C++ has a richer set of (standard) libraries than C, but a consequence of the previous point is that C++ libraries often use features that are inappropriate for small systems and are hence not uasble on small systems.
- C++ has more constructs than C that allow you to shoot yourself in the foot.
- C++ has more constructs than C that allow you to *prevent* yourself from shooting yourself in the foot (yes, IMO this and the previous one are both true).
- C++ has a richer set of abstraction mechanisms, so it enables better ways of programming, especially for libraries.
- C++ language features (for instance constructors/destructors, conversion functions) make it more difficult to see through the code to see the generated machine and thus the cost in space and time of a language construct.
- C++ language construct make it less necessary to be aware of how exactly the are translated to machine code because they do 'the right thing' in a more abstract way.
- The C++ language standard is evolving quickly and is adopted speedily by the big compilers (gcc, clang, microsoft). C is evolving rather sowly, and adoption of some newer features (variant arrays) is scare and has even been reverted in a later standard. This point in particular is interesting in that different people use it to support the opposite positions.
- C++ is undoubtedly a sharper tool than C. Do you trust your programmers (or yourself) to use such a tool to make a beautiful sculpture, or do you fear them hurting themselves and would you rather settle for a less beautiful but lower-risk product? (I recall that my sculpture teacher once told me that blunt tools can in some situations be *more* dangerous than sharp ones.)

My blog has some writings on using C++ on small systems (= micro-controllers).

edited Jun 3 '15 at 10:20 community wiki
4 revs
Wouter van Ooijen

My background: just out of school training under old Bell Labs programmers; been working for 3 years, 2 on undergrad research project; data acquisition / process control in VB.NET. Spent 1.5 years doing work on an enterprise database application in VB6. Currently working on project for *embedded* PC with 2GB of storage, 512MB of RAM, 500MHz x86 CPU; several apps running concurrently written in C++ with an IPC mechanism in between. Yes, I'm young.

My opinion: I think C++ can work effectively *given the environment I've written above*. Admittedly, hard real-time performance isn't a requirement for the app I'm on, and in some embedded applications, that can be an issue. But here are the things I've learned:

- *C++ is fundamentally different than C* (ie, there is no C/C++). While everything that is valid C is valid C++, C++ is a very different language and one needs to learn how to program in C++, not C, to effectively use it in *any* situation. In C++, you need to program object-orientedly, not procedurally, and not a hybrid of the two (big classes with lots of functions). In general, you should focus on making small classes with few functions, and compose all the small classes together into a larger solution. One of my coworkers explained to me that I used to program procedurally in objects, which is a grand mess and is hard to maintain. When I started to apply more object-oriented techniques, I found my code's maintainability/readability went up.
- C++ provides additional features in the form of object-oriented development that can provide *a way to simplify code to make it easier to read/maintain*. Honestly, I don't think there's much in the way of a performance/space efficiency improvment in doing OOP. But I think OOP is a technique that can help split up a complex problem into lots of little pieces. And that is helpful for the people working on the code, an element of this process that should not be ignored.
- Many arguments against C++ have primarily to do with dynamic memory allocation. C has this same problem too. You can write an object oriented application without using dynamic memory, although one of the benefits of using objects is that you can allocate these things dynamically in an easy fashion. Just as in C, you have to be careful about how manage the data to reduce memory leaks, but the RAII technique makes this simpler in C++ (make dynamic memory destruct automatically by encapsulating it in objects). In some applications, where every memory location counts, this may be too wild & wooly to manage.

EDIT:

- *WRT the "Arduino C++" question*: I would argue that C++ without dynamic memory management can *still* be useful. You can organize your code into objects, and then place those objects into various locations within your application, setup callback interfaces, etc. Now that I have been developing in C++, I can see many ways in which an application with all data allocated on the stack can still be useful with objects. I will admit though - I never actually written an embedded app like that for the Arduino, so I have no proof behind my claim. I have some opportunities to do some Arduino development in an upcoming project - hopefully I can test my claim there.

edited Nov 3 '10 at 17:33 — community wiki, 3 revs, J. Polfer

2 I would like to comment on your second point, you say that it helps break up a complex problem into a lot of little pieces and that feature should be ignored. This is the exact reason that I am so pro-C++. A very large body of research into programming shows that a linear growth in program size gives an exponential growth in development time. this follows the opposite way, if you can properly split up a program then you can give an exponential decay in development time. This is by far the most important thing. – Kortuk ♦ Jun 16 '10 at 14:49

on your second point as well: Simply using an OOP design methodology doesn't produce more compartmentalized code. Having a good base design does, how one expresses that design is left to the developer. OOP does not define that you separate your code properly, it provides another option, and more over, the appearance that you did so, but, it certainly does not enforce good design, that is up to the developer. – Mark Jun 17 '10 at 2:58

That is always true. I have never heard of a language that does enforce good design. I think we mostly imply that is the developers job and that C++ makes it easy to use and implement in an organized fashion. – Kortuk ♦ Jun 17 '10 at 4:43

@Mark - I agree. It has been a learning process for me. – J. Polfer Jun 17 '10 at 17:13

Yes, the issue with C++ is the increased footprint of the code.

In some systems you are counting bytes, and in that case you are going to have to accept a cost of running that close to the bounds of your systems are increased development cost of C.

But, even in C, for a well designed system you need to keep everything encapsulated. Well designed systems are hard, and C++ give programmers a place for a very structured and controlled method of development. There is a cost to learn OOP, and if you want to switch to it you much accept it, and in many cases the management would rather continue with C and not pay the cost, as it is hard to measure the results of a switch that increases productivity. You can see an article by embedded systems guru Jack Ganssle here.

Dynamic memory management is the devil. Not really, the devil is auto-route, dynamic memory management works great on a PC, but you can expect to restart a PC every few weeks at least. You will find that as an embedded system continues to run for 5 years that dynamic memory management can really get screwed up and actually start failing. Ganssle discusses things like stack and heap in his article.

There are some things in C++ that are more prone to causing problems and use many resources, removing dynamic memory management and templates are big steps to keep the footprint of C++ closer to the footprint of C. This is still C++, you do not need dynamic memory management or templates to write good C++. I did not realize they removed exceptions, I consider exceptions an important part of my code that I remove in the release, but use until that point. In field testing I can have exceptions generate messages to inform me of an exception being caught.

edited Jun 16 '10 at 18:11 — community wiki, 2 revs, Kortuk

1 I used to agree that code footprint is a problem, but recently it seems that flash size has very litte influence on the price of a microcontroller, much much less than RAM size or the number of IO pins. – Wouter van Ooijen Sep 6 '11 at 13:21

The argument on dynamic memory is more important IMO. I've seen industrial systems which could run for weeks non-stop, but the diagnostic layer (written in C++) would limit the time to restart to some 12 hours. – Dmitry Grigoryev Jun 3 '15 at 12:57

I thought this anti-C++ rant by Linus Torvalds was interesting.

> One of the absolute worst features of C++ is how it makes a lot of things so context-dependent - which just means that when you look at the code, a local view simply seldom gives enough context to know what is going on.

He's not talking about the embedded systems world, but Linux kernel development. To me, the relevance comes from this: C++ requires understanding a larger context, and I can learn to use a set of object templates, I don't trust myself to remember them when I have to update the code in a few months.

(On the other hand, I'm currently working on an embedded device using Python (not C++, but using the same OOP paradigm) that will have exactly that problem. In my defense, it's an embedded system powerful enough to be called a PC 10 years ago.)

answered Jun 15 '10 at 16:52 — community wiki, pingswept

3 We may differ, but I find that just opening any project I cannot tell what is going on immediately, but if I know something about what it is doing and I have something well coded in C and something well coded in C++ the C++ always seems more clear. You still need to implement encapsulation for good development in C, which C++ makes very easy to do. Proper use of classes can make it very clear where you interfaces are, and they can be completely handled through an object. – Kortuk ♦ Jun 15 '10 at 18:54

Totally agreed on encapsulation and classes. Operator overloading and inheritance, not so much. – pingswept Jun 15 '10 at 19:26

1 Haha, yes, operator overloading can be used to obfuscate the function of code. If someone is operator overloading it needs to be for clear reasons or not done at all. Inheritance should only be used in specific cases where you actually are doing something that is just like the parent with a few additions. I think I would not every use either function in OOP. I have used both, but in an embedded system cannot think of a case were I would. Just as I think a compiler with an 80 character limit on variable names should be immediately scrapped. – Kortuk ♦ Jun 15 '10 at 19:53

2 I just threw up in my mouth a little at the thought of programming an MCU in Python... – vicatcu Jun 15 '10 at 20:14

You are not the only one, but if it works well and is efficient, I can forgive. – Kortuk ♦ Jun 15 '10 at 21:01

I think other answers made a pretty good case for the pros and cons and decision factors, so I'd like just to summarize and add a few comments.

For small microcontrollers (8-bit), no way. You're just asking to hurt yourself, there's no gain and you'll give up too much resources.

For high-end microcontrollers (e.g. 32-bit, 10s or 100s of MB for RAM and storage) that have a decent OS it's perfectly OK and, I'd dare to say, even recommended.

So the question is: where's the boundary?

I don't know for sure, but once I developed a system for a 16-bit uC with 1 MB RAM & 1 MB storage in C++, only to regret it later. Yes, it worked, but the extra work I had wasn't worth it. I had to make it fit, make sure things like exceptions wouldn't produce leaks (the OS+RTL support was pretty buggy and unreliable). Moreover, an OO app typically does lots of small allocations, and the heap overhead for those was another nightmare.

Given that experience, I'd assume for future projects that I'll choose C++ only in systems at least 16-bit, and with at least 16 MB for RAM & storage. That's an arbitrary limit, and probably will vary according to things like the type of application, coding styles and idioms, etc. But given the caveats, I'd recommend a similar approach.

edited Jan 12 '11 at 14:41 community wiki
2 revs
fceconel

1 I have to disagree here, the is no sudden point where C++ becomes acceptable due to system resources, good design practice can keep C++'s footprint where C's footprint is. This results in code with OOP designs that take the same space. Poorly written C can be just as bad. – Kortuk ♦ Jan 12 '11 at 15:35

1 Well, it depends on how big your application is and how much you use certain features that require more space (like templates and exceptions). But personally I'd rather use C than have to limit myself to a restrained C++. But even then you'll have the overhead of a bigger RTL, virtual method thunks, constructor/destructor chain invocation... these effects may be mitigated with careful coding, but then you're losing the main reason for C++ use, abstraction and high level perspective. – fceconel Jan 12 '11 at 18:12

There are some features of C++ which are useful in embedded systems. There are others, like exceptions, which can be expensive, and whose costs may not always be apparent.

If I had my druthers, there would be a popular language which combined the best of both worlds, and included some features which are lacking in both languages; some vendors include a few such features, but there are no standards. A few things I'd like to see:

1. Exception handling a little more like Java, where functions which can throw or leak exceptions must be declared as such. While a requirement for such declarations may be somewhat annoying from a programming perspective, it would improve the clarity of code in cases where a function may return an arbitrary integer if it succeeds, but may also fail. Many platforms could handle this inexpensively in code by e.g. having the return value in a register and success/failure indication in the carry flag.
2. Overloading of static and inline functions only; my understanding is that the standards bodies for C have avoided function overloading so as to avoid a need for name mangling. Allowing overloads of static and inline functions only would avoid that problem, and would give 99.9% of the benefit of overloading external functions (since .h files could define inline overloads in terms of differently-named external functions)
3. Overloads for arbitrary or specific compile-time-resolvable constant parameter values. Some functions may inline very efficiently when passed with any constant value, but inline very poorly if passed a variable. Other times code which may be an optimization if a value is constant may be a pessimization if it isn't. For example:

```
inline void copy_uint32s(uint32_t *dest, const uint32_t *src, __is_const int n)
{
  if (n <= 0) return;
  else if (n == 1) {dest[0] = src[0];}
  else if (n == 2) {dest[0] = src[0]; dest[1] = src[1];}
  else if (n == 3) {dest[0] = src[0]; dest[1] = src[1]; dest[2] = src[2];}
  else if (n == 4) {dest[0] = src[0]; dest[1] = src[1]; dest[2] = src[2]; dest[3] =
src[3];}
  else memcpy((void*)dest, (const void*)src, n*sizeof(*src));
}
```

   If 'n' can be evaluated at compile time, the above code will be more efficient than a call to memcpy, but if 'n' can't be evaluated at compile time the generated code would be much bigger and slower than code which simply called memcpy.

I know the father of C++ isn't too keen on an embedded-only version of C++, but I would think it could offer some considerable improvements over just using C.

Anyone know if anything like the above are being considered for any type of standard?

edited May 13 '11 at 16:06 community wiki
2 revs
supercat

en.wikipedia.org/wiki/Embedded_C%2B%2B – Toby Jaffey May 13 '11 at 15:54

@Joby Taffey: I guess I edited my post to omit mention that the creator of C++ wasn't keen on an embedded subset; I'm aware that there were efforts, but by my understanding they hadn't really gotten all that far. I do think there's would be definite use for a standardized language which would be amenable to 8-bit processors, and features such as I described above would seem useful on any platform. Have you heard of any languages offering things like #3 above? It would seem very useful, but I've never seen any language offer it. – supercat May 13 '11 at 16:05

"The father of C++" has zero experience of embedded systems programming, so why would anyone care about his opinion? – Lundin Jun 4 '15 at 6:38

@Lundin: The fact that some influential people do seem to care about his opinions on various matters would seem to be reason, in and of itself, for other people to do so. I'm thinking that since I wrote the above the increasing power of templates may have added new possibilities for having overloads based upon what constants can be resolved at compile-time, though far less cleanly than if such a thing were supported as a compile-time feature (from what I understand, one would specify a template which should try various things in order and go with the first that doesn't fail... – supercat Jun 4 '15 at 13:33

...but that would require the compiler to waste a fair bit of effort compiling potential substitutions which would then end up being discarded. Being able to more cleanly say "If this is a constant, do this; otherwise do this" without any "false starts" would seem a cleaner approach. – supercat Jun 4 '15 at 13:34

I hope to add more light than heat to this discussion about C++ on bare metal and resource constrained systems.

Problems in C++:

- Exceptions are especially a RAM problem as the required "emergency buffer" (where the out of memory exception goes for example) can be larger than the available RAM and is certainly a waste on microcontrollers. For more info see n4049 and n4234. They should be turned off (which is currently unspecified behavior so be sure and not ever throw). SG14 is currently working on better ways of doing this.
- RTTI is probably never worth the overhead, it should be turned off
- Large debug builds, although this is not a problem in classic desktop development if the debug does not fit on the chip it can be a problem. The problem arises from templated code or extra function calls added for clarity. These extra function calls will be removed again by the optimizer and the added clarity or flexibility can be a great advantage, however in debug builds this can be a problem.
- Heap allocation. Although the STL allows for the use of custom allocators this can be complex for most programmers. Heap allocation is non deterministic (i.e. not hard real time) and fragmentation can lead to unexpected out of memory situations to occur despite having worked in testing. The book keeping needed by the heap in order to keep track of free space and varying sized can be a problem with small objects. Its usually better to use pool allocation (both in C and C++) but this can be abnormal for C++ programmers used to only using the heap.
- Runtime polymorphism and other indirect calls are usually a big performance hit, the problem is usually more because the optimizer cannot see through them more than the actual fetching and jumping to the address. Indirect calls are to be avoided for this reason in C and C++ where as in C++ they are more ingrained in the culture (and are quite useful in other domains).

In conclusion C++ has some problems but they are essentially all fixable.

Now for C, here the problem is higher order. I do not have the syntactic ability in C to abstract things in a way that I can perform optimization or check invariants at compile time. Therefore I cannot properly encapsulate things in a way that the user does not need to know how they work in order to use them and most of my error detection is done at runtime (which is not only too late but also adds cost). Essentially the only way to be generic in C is through data, I pass a format string to printf or scanf which is evaluated at runtime for example. It is then quite hard for the compiler to prove that I am not using some of the options which are theoretically possible when passed the right data which means potential dead code generation and loss of optimization potential.

I know I may be unleashing a shitstorm here but my experience on 32 bit microcontrollers is that in an apples to apples comparison of C and C++ both written by experts (as in C++ potentially highly templated) C++ is the much more efficient language as soon as anything needs to be at all generic (as in any library) and they are essentially equivalent in non generic cases. It is also easier for a novice to leverage the expertise of an expert library implementer in C++.

At the same time there are actually truly few functions to which I cannot pass incorrect data, as soon as the input is not an int but a `something` for which I happen to be using an int as a method of representation then there is a potential get it wrong (pass an invalid value or an 'otherThing' rather than a 'something'). In C my only method of checking if the user got it wrong is at runtime. In C++ I have the ability to perform some checks, not all checks but some checks at compile time which are free.

At the end of the day a C team is often as powerful as its weakest programmer and the resulting code's benefit has either a multiplayer of 1 or a performance penalty. What I mean by this is it is either high performance for one and only one unique job in a unique environment of unique design decisions or it is generic enough to be used in multiple environments (other microcontroller, other memory management strategy, other latency vs. throughput trade offs etc. etc.) but has an inherent performance cost.

In C++ things can be encapsulated by experts and used in many environments where compile time code generation adapts to the specific task and static checking keeps users from doing stupid stuff at zero cost. Here we have far less trade off between being generic and being fast and thus ultimately from an cost vs. benefit standpoint are the more performant, safer and more productive language.

It is a valid critique that there is still a great shortage of good C++ libraries for embedded, this can lead to pragmatic decisions to use mostly C on a C++ compiler. Decisions to use only C in a project are essentially either ideologically driven, out of need for legacy support or an admission that the team is not disciplined enough to refrain from a very select set of stupid things which one can do in C++ but not in C and at the same time disciplined enough to not do any of the far greater set of stupid things which one cannot guard against in C but could in C++.

answered 19 hours ago — community wiki odinthenerd

Nice addition to my answer :) Who would this mysterious C++ lover be? His profile states "Apparently, this users prefers to keep an air of mystery about them." (bad english, BTW) BUT AHA the location is "Bochum, Germany"..... See you at the conference! – Wouter van Ooijen 17 hours ago

Ah yeah updated my profile ;) nice to know your coming to emBO++ it will be a good crowd – odinthenerd 16 hours ago

My background, embedded (mcu, pc, unix, other), realtime. Safety critical. I introduced a previous employer to STL. I don't do that anymore.

Some Flame content

Is C++ suitable for embedded systems?

Meh. C++ is a pain to write and a pain to maintain. C+ is sort-of okay (don't use some features)

C++ in Microcontrollers? RTOSes? Toasters? Embedded PCs?

Again I say Meh. C+ is not too bad, but ADA is less painful (and that's really saying something) . If you're lucky like me, you get to do embedded Java. Checked array access and no pointer arithmetic makes for very reliable code. Garbage collectors in embedded Java are not highest priority, and there is scoped memory and object reuse, so well designed code can run forever without a GC.

Is OOP useful on microcontrollers?

Sure is. The UART is an object..... The DMAC is an object...

Object State machines are very easy.

Does C++ remove the programmer too far from the hardware to be efficient?

Unless it's a PDP-11, C ain't your CPU. C++ was originally a preprocessor ontop of C so Bjarne Stroustrup would stop getting laughed at for having slow Simula simulations while at AT&T. C++ ain't your CPU.

Go get an MCU that runs java bytecodes. Program in Java. Laugh at the C guys.

Should Arduino's C++ (with no dynamic memory management, templates, exceptions) be considered as "real C++"?

Nope. just like all the bastardised C compilers out there for MCU's.

Forth, Embedded Java or Embedded ADA are standardised(ish); all else is sorrow.

edited Dec 15 '12 at 23:25 — community wiki 2 revs Tim Williscroft

2 Is it that easy to find microcontrollers supporting Java? I'd think this would limit the choices considerably. And what are your experiences about performance penalty (since in uCs you usually wouldn't have JIT)? What about the impact of GC unpredictability in realtime systems? – fceconel Jan 12 '11 at 14:39

2 What MCUs are out there that support embedded Java? – J. Polfer Jan 12 '11 at 17:36

www.ajile.com for starters. – Tim Williscroft Jan 13 '11 at 9:41

+1 for Ada. It's got a lot going for it in embedded, including Arduinos. – Brian Drummond Dec 15 '12 at 14:14

portable java VM for micros written in c is open source. dmitry.co/index.php?p=./04.Thoughts/... – Tim Williscroft Dec 15 '12 at 23:28

C++ is more that one programming language:

a) It's a "better" C b) It's an object oriented language c) It's a language that allows us to write generic programs

Although all of these features can be used separately the best results are achieved when the three of them are used at the same time. Nonetheless, if you choose to pick just one of them the quality of the embedded software will increase.

a) It's a "better" C

C++ is a strong typed language; stronger than C. Your programs will benefit from this feature.

Some people is afraid of pointers. C++ includes the references. Overloaded functions.

And worth to say: None of these features incurred in bigger or slower programs.

b) It's an object oriented language

Someone said in this post that abstracting the machine in microcontrollers is not a good idea. Wrong! All of us, the embedded engineers, have always abstracted the machine, just with other sintax that that of C++. The problem I see with this argument is that some programmers are not used to think in objects, that's way they don't see the benefits of OOP.

Whenever you are ready to use a microcontroller's peripheral it's likely that the peripheral has been abstracted for us (from yourself or a third party) in the form of the device driver. As I said before, that driver uses the C sintax, as the next example shows (taken directly from a NXP LPC1114 example):

```
/* Timer setup for match and interrupt at TICKRATE_HZ */

Chip_TIMER_Reset(LPC_TIMER32_0);

Chip_TIMER_MatchEnableInt(LPC_TIMER32_0, 1);

Chip_TIMER_SetMatch(LPC_TIMER32_0, 1, (timerFreq / TICKRATE_HZ2));

Chip_TIMER_ResetOnMatchEnable(LPC_TIMER32_0, 1);

Chip_TIMER_Enable(LPC_TIMER32_0);
```

Do you see the abstraction? So, when using C++ for the same purpose, abstraction is brought to the next level through abstraction and encapsulation mechanism of C++, at zero cost!

c) It's a language that allows us to write generic programs

Generic programs are achieved through templates, and templates also have no costs for our programs.

Besides, static polymorphism is achieved with templates.

Virtual methods, RTTI and exceptions.

There is a compromise when using virtual methods: better software vs some penalty in performance. However, remember that dynamic binding is likely to be implemented using a virtual table (an array of function pointers). I have done the same in C a lot of times (even in a regular basis), so I don't see the drawbacks in using virtual methods. Moreover, virtual methods in C++ are more elegant.

Finally, an advice about RTTI and exceptions: DON'T USE THEM in embedded systems. Avoid them at all cost!!

answered Jun 27 '14 at 17:31 community wiki fjrg76

Embedded systems are designed to do some specific task, rather than be a general-purpose computer for multiple tasks. An embedded system is a combination of computer hardware and software. C is the mother of all the modern languages. It’s a low level but power full language and deals all kind of hardware. So C/C++ is an optimal choice for developing software for embedded system, which is very use full for every embedded system .As we know C is a developing language. The operating system UNIX is written in C .Because successful software development is so frequently about selecting the best language for a given project, it is surprising to

find that C/C++ language has proven itself appropriate for both 8-bit and 64-bit processors; in systems with bytes, kilobytes, and megabytes of memory. C has the benefit of processor-independence, which allows programmers to concentrate on algorithms and applications, rather than on the details of particular processor architecture. However, many of these advantages apply equally to other high-level languages. But C/C++ succeeded where so many other languages have largely failed?

edited Jun 29 '12 at 12:44 — community wiki
2 revs, 2 users 67%
IMRAN AHMAD AWAN

6 I'm really not sure what this adds to the discussion. – Dave Tweed ♦ Oct 19 '12 at 22:10

<rant>

I think C++ is a crappy language in the first place. If you want to use OOP, write Java programs. C++ does nothing to enforce OOP paradigms, as direct memory access is fully within your power to (ab)use.

If you have an MCU, you're talking about most likely less than 100kB of flash memory. You want to be programming in a language whose abstraction of memory is: when I declare a variable or an array, it gets memory, period; malloc (aka "new" keyword in C++) should be more or less banned from use in embedded software, except perhaps in rare occasions one call during program startup.

Hell, there are (frequently) times in embedded programming where C is not quite low-level enough, and you need to do things like allocate variables to registers, and write inline assembly to tighten up your interrupt service routines (ISRs). Keywords like "volatile" become pretty darn important to understand. You spend a lot of your time manipulating memory at the *bit* level, not the *object* level.

Why would you want to delude yourself into thinking that things are simpler than they in fact are?

</rant>

answered Jun 15 '10 at 20:21 — community wiki
vicatcu

My issue here is simply, why do I want to know the complexity of the driver that was written to control USART1 if it has been fully developed to handle the interface. – Kortuk ♦ Jun 15 '10 at 21:19

1 I did not down vote you, but I would like to point out that C++ does not need to enforce OOP, just gives the tools to do it. Enforcing good coding standards is the job of the developer. It can help if the language makes it easier, but the language will never do it on it's own. C can be unreadable in some cases. – Kortuk ♦ Jun 16 '10 at 18:09

1 All languages are good for something. C++ is fast. OOP, if well done, makes it much easier for multiple developers to work in parallel and to code for the unknown. I think this is why it has so much traction in game development. – Toby Jaffey Jun 17 '10 at 23:08

1 Yes, I agree. The reason I see it for the embedded world is because of the sheer amount of features and functions being added to alot of the different systems already in place and new systems being developed. Project get bigger and bigger. Either we take longer to develop them or we start applying and contorting what the CS world has already done on PCs. – Kortuk ♦ Jun 18 '10 at 20:51

4 Yet another person who doesn't understand C++ properly. It always amazes me how many there are. – Rocketmagnet Feb 22 '12 at 22:34

protected by Nick Alexeev ♦ Jun 10 '15 at 23:18

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 reputation on this site (the association bonus does not count).

Would you like to answer one of these unanswered questions instead?