embedded
cracking the code to systems development
(http://www.embedded.com)

CONTACT US (/CONTACTUS) • SUBSCRIBE TO NEWSLETTERS

login register

DEVELOPMENT ESSENTIALS & EDUCATION COMMUNITY ARCHIVES ABOUT US Search

Home (/) > Programming Languages & Tools Development Centers (/development/programming-languages-and-tools) > Design How-To (/development/programming-languages-and-tools/articles)

A guide to C++ for C programmers

Mark Kraeling, GE Transportation

NOVEMBER 15, 2013

Share 14 G+1 24 Tweet

(mailto:?subject=A guide to C++ for C programmers&body=http://www.embedded.com/design/programming-languages-and-tools/4424383/A-guide-to-C--for-C-programmers)

Editor's Note: *A bare bones guide to the C++ language for C programmers, excerpted from* ***Software engineering for embedded systems (http://www.elsevier.com/books/software-engineering-for-embedded-systems/oshana/978-0-12-415917-4)*** *by Mark Kraeling.*

There are a number of reasons developers may want to consider using C++ as the programming language of choice when developing for an embedded device. C++ does compare with C in terms of syntactical similarities, in addition to memory allocation, code reuse and other features. There are also reasons to take caution when considering C++ and its related toolsets.

One reason is that functionality and performance vary across compilers due to differing implementations of the standard by individual vendors and open-source offerings. In addition, C++ and its libraries tend to be much larger and more complex than their C language counterparts. As such, there tends to be a bit of ambiguity in the community around C++ as a viable option for embedded computing, and more specifically what features of the language are conducive to embedded computing and what features should generally be avoided.

When characterizing the cost of various aspects of using C++ for embedded software development, we characterize cost as something that requires runtime resources. Such resources may be additional stack or heap space, additional computational overhead, additional code size or library size, etc. When something can be done offline a priori by the compiler, assembler, linker or loader, we consider those features to be inexpensive and in some cases absolutely free.

As behaviors differ across compilers and vendors, the burden is ultimately placed on the developer and designer to ensure that said benefits are actually achieved with a given development environment for the target architecture. Lastly, development tools change over time as new functionality is added, features are deprecated, performance is tuned and so forth.

Development tools are highly complex interdependent software systems, and as such there may periodically be regressions in performance of legacy software as tools evolve. A periodic re-evaluation of features and performance is encouraged. The topics discussed in this section are furthermore presented as general trends, and not meant to be an absolute for any specific target or toolset implementation.

MOST READ

11.01.2013
Inline Code in C and C++ (/design/programming-languages-and-tools/4423679/Inline-Code-in-C-and-C-)

11.08.2013
How to know when to switch your SCM/version control system (/design/programming-languages-and-tools/4424039/How-to-know-when-to-switch-your-SCM-version-control-system)

10.26.2013
ARM design on the mbed Integrated Development Environment - Part 1: the basics (/design/programming-languages-and-tools/4423344/ARM-design-on-the-mbed-Integrated-Development-Environment---Part-1--the-basics)

EMBEDDED (/VIDEOS) TV (/VIDEOS) (/videos) video library (/videos)

Relatively inexpensive features of C++ for embedded

In the following section I detail C++ language features that are typically handled automatically by the compiler, assembler, linker and/or loader effectively free. That is to say they typically will not incur additional computational or storage overhead at run-time, or increase code size.

Static constants. C++ allows users to specify static constants in their code rather than use C-style macros. Consider the example below:

C language:
```
#define DRIVE_SHAFT_RPM_LIMITER 1000
```
C++ language:
```
const int DRIVE_SHAFT_RPM_LIMITER 5 1000
```

Developers may take pause in that the C++ language implementation will require additional storage space for the variable DRIVE_SHAFT_RPM_LIMITER. It is the case, however, that if the address of said variable is not used within the code and rather the literal value 1000 is used in computation, the compiler will fold in the value as a constant at compilation time, thus eliminating the storage overhead.

Ordering of declarations and statements. In the C programming language, programmers are required to use a specific sequence whereby blocks start with declarations followed by statements. C++ lifts this restriction, allowing declarations to be mixed in with statements in the code. While this is mostly a syntactical convenience, developers should also use caution regarding the effect on the readability and maintainability of their code.

Function overloading. Function overloading pertains to the naming conventions used for functions, and the compiler's ability to resolve at compile time which version of a function to use at the call site. By differentiating between various function signatures, the compiler is able to disambiguate and insert the proper call to the correct version of the function at the call site. From a run-time perspective, there is no difference.

Usage of namespaces. Leveraging code reuse has the obvious benefits of improving reliability and reducing engineering overhead, and is certainly one promise of C++. Reuse of code, especially in the context of large software productions, often comes with the challenge of namespace collisions between C language functions depending on how diligent past developers have been with naming convention best practices. C++'s classes help to avoid some of these collisions, but not everything can be constructed as a class (see previously); furthermore existing C language libraries must still be accommodated in many production systems.

C++'s namespaces resolve much of this problem. Any variables within the code are resolved to a given namespace, if nothing else the global namespace. There should be no penalty in using these name spaces for organizational advantage.

Usage constructors and destructors. C++ adds the functional of "new" and "delete" operators for provisioning and initializing heap-based objects. It is functionally equivalent to using malloc and initialization in C, but has the added benefit of being easier to use and less prone to errors in a multi-step allocation and initialization process.

C++'s "delete" functionality is also similar to "free" in C; however, there may be run-time overhead associated with it. In the case of C, structs are not typically destructed like objects in C++. Default destructors in C++ should be empty, however. One caveat with new/delete is that certain destructors may throw run-time exceptions which would in turn incur overhead. Run-time exceptions are described in more detail in the following subsections.

Modestly expensive features of C++ for embedded

The following groups of features do not necessarily need to impact the program run-time versus their C programming counterparts, but in practice they may have an effect depending on maturity and robustness of the compiler and related tools.

Inlining of functions. The subject of inlining functions for C++ is a very broad one, with far-reaching performance impacts ranging from run-time performance to code size and beyond. When designating a function to be inlined, typically the "inline" keyword is used.

Some compilers will take this as a hint, while others will enforce the behavior. There may be other pragmas available within a given toolset for performing this action in a forceable manner, and documentation should be revisited accordingly. One of the costs associated with function inlining is naturally growth in code size, as, rather than invoke the function via a call site at run-time, the compiler inserts a copy of the function body directly where the call site originally was.
Additionally, there may be performance impacts due to challenges in register allocation across procedure boundaries or increase in register pressure within the calling function. It is advised to closely consider the impact of inlining for your target when using C++.

MOST COMMENTED

02.17.2015

Modern C++ in embedded systems – Part 1: Myth and Reality (/design/programming-languages-and-tools/4438660/Modern-C--in-embedded-systems---Part-1--Myth-and-Reality)

RELATED CONTENT

10.12.2011 | TECHNICAL PAPER

How to use C++ Model effectively in SystemVerilog Test Bench (/electrical-engineers/education-training/tech-papers/4230525/How-to-use-C-Model-effectively-in-SystemVerilog-Test-Bench)

06.30.2008 | DESIGN

Dynamic allocation in C and C++ (/design/real-time-and-performance/4007614/Dynamic-allocation-in-C-and-C-)

05.07.2007 | TECHNICAL PAPER

C++ Under the Hood (/electrical-engineers/education-training/tech-papers/4126302/C--Under-the-Hood)

07.02.2009 | TECHNICAL PAPER

Recursion C++ DSP Toolkit: DM6437 Cross Development (/electrical-engineers/education-training/tech-papers/4137772/Recursion-C--DSP-Toolkit-DM6437-Cross-Development)

06.29.2010 | TECHNICAL PAPER

The Inefficiency of C++, Fact or Fiction? (/electrical-engineers/education-training/tech-papers/4200968/The-Inefficiency-of-C--Fact-or-Fiction-)

PARTS SEARCH

Datasheets.com (http://www.datasheets.com)

Constructors, destructors and data type conversions. If a developer does not provide constructors and destructors for a given C++ class, the compiler will automatically provision for them. It is true that these default constructors and destructors may not ever be required; moreover the developer may have explicitly omitted them, as they were not required.

Dead-code elimination optimizations will likely remove these unused constructors and destructors, but care should be taken to ensure this is in fact the case. One should also take caution when doing various copy operations and conversion operations: for example, passing a parameter to a member function by value, in which a copy of the value must be created and passed using the stack. Such scenarios may inadvertently lead to invocation of constructors for the value being copied, which subsequently cannot be removed by dead-code elimination further on in the compilation process.

Use of C++ templates. The use of templates within C++ code for embedded systems should come with no overhead, as in principle all of the work is done ahead of time by the build tools in instantiating the right templates based on source code requirements. The parameterized templates themselves are converted into non-parameterized code by the time it is consumed by the assembler. In practice, however, there have been cases of compilers that behave in an overly conservative (or aggressive, depending on your view point) manner, and instantiate more template permutations than were required by the program. Ideally dead-code elimination would prune these out, but that has been shown to not always be the case on some earlier C++ compilers.

Share 14 G+1 24 Tweet

(mailto:?subject=A guide to C++ for C programmers&body=http://www.embedded.com/design/programming-languages-and-tools/4424383/A-guide-to-C--for-C-programmers)

< Previous Page 1 of 2 Next > (/design/programming-languages-and-tools/4424383/2/A-guide-to-C--for-C-programmers)

WRITE A COMMENT

Subscribe to RSS updates all articles (/rss/all) or category ▾

DEVELOPMENT CENTERS

All Articles (/development)

Configurable Systems (/development/configurable-systems)

Connectivity (/development/connectivity)

Debug & Optimization (/development/debug-and-optimization)

MCUs, Processors & SoCs (/development/mcus-processors-and-socs)

Operating Systems

ESSENTIALS & EDUCATION

Products (/products/all)

News (/news/all)

Source Code Library (/education-training/source-codes)

Webinars (/education-training/webinars)

Courses (/education-training/courses)

Tech Papers (/education-training/tech-papers)

COMMUNITY

Insights (/insights)

Forums (http://forums.embedded.com)

Events (/events)

ARCHIVES

Embedded Systems Programming / Embedded Systems Design Magazine (/magazines)

Newsletters (/archive/Embedded-com-Tech-Focus-Newsletter-Archive)

Videos (/videos)

Collections (/collections/all)

ABOUT US

About Embedded (/aboutus)

Contact Us (/contactus)

Newsletters (/newsletters)

Advertising (/advertising)

Editorial Contributions (/editorial-contributions)

Site Map (/site-map)

(/development
/operating-systems)

Power Optimization
(/development/power-
optimization)

Programming
Languages & Tools
(/development
/programming-
languages-and-tools)

Prototyping &
Development
(/development
/prototyping-
and-development)

Real-time &
Performance
(/development/real-
time-and-performance)

Real-world Applications
(/development/real-
world-applications)

Safety & Security
(/development/safety-
and-security)

System Integration
(/development/system-
integration)