

Embedded Systems

PROGRAMMING

by P.J. PLAUGER

Embedded C++: An Overview

Is EC++ the diet pill that will shrink C++ down to size for embedded applications? Here's P.J. Plauger's take.

The Fall 1997 Embedded Systems Conference in San Jose was a smashing success. Attendance was way up over last year, and rather more than even the optimistic projections made by the Powers that Be before the conference opened. I haven't seen this much activity and interest in the embedded marketplace ever before. Period.

Of particular interest to me was the attention awarded to Embedded C++. It was only a year earlier, at the Fall 1996 Embedded Systems Conference, that I had the honor of formally introducing this project to the public. It could have sunk without a trace in the intervening months—many optimistic new projects do—but that didn't happen. Quite the con-

trary, I found growing enthusiasm on several fronts.

- The Embedded C++ Technical Committee, a consortium of Japanese companies, held their third international meeting on September 30, across the street from the convention center. About 50 people showed up, many of them vendors announcing new products that support this de facto industry standard. Several major Japanese companies have also joined the original four semiconductor heavy-hitters as well
- I repeated my talk on Embedded C++ on October 1 to an audience of at least 150 people. There was obvious interest and lots of intelligent questions

P.J. Plauger is the author of the standard C++ library shipped with Microsoft Visual C++. His "State of the Art" column appears monthly in Embedded Systems Programming.

- Green Hills Software is now marketing their embedded compiler products with full EC++ support
- Metrowerks and Hiware, two other companies that license compilers into the embedded marketplace, announced their intention to supply compilers that support Embedded C++ in the near future
- I talked to half a dozen exhibitors on the show floor who already knew about Embedded C++ and were beginning to plan how to support it for their customers

I'll elaborate on these themes in the remainder of this article. But first, let me back up and supply a bit of history. I can then fill in the relevant technical details. (See also the sidebar, "Read All About It," for a supplemental reading list.)

HISTORY

I first heard of this project in November 1995, while attending a meeting of WG21 and J16, the ISO and ANSI committees standardizing C++. Advanced Data Controls Corp. (ADaC) is a Japanese company with whom I've done business since their founding in 1982. My friends at ADaC urged me to extend my stay to attend a Friday evening meeting they had arranged. Given the warmth and past success of our business relationship, I didn't hesitate to rearrange my travel plans and attend.

The meeting was astonishing. In addition to the folks from ADaC and a couple of fellow Americans, the attendees included software people from the four major Japanese semiconductor manufacturers—Fujitsu, Hitachi, NEC, and Toshiba. These were the folks responsible for providing software development tools to their customers. They were all wrestling with what to do about C++, and they were very frank about their problems. I have personally never witnessed such an open dialog between nominal competitors, certainly not among American companies. But then, we Americans enjoy a richer fabric of antitrust laws, to keep companies from collaborating unfairly. Or fairly, for that matter.

Moving a C++ program between compilers has become more difficult, not less, as a result of standardization.

All four companies shared a common complaint. Their customers—almost all developers of embedded systems using their chips—were beginning to demand C++ compilers. Naturally, the vendors were eager to oblige. All four companies offered some form of a C++ compiler, or were about to. But when customers started using C++, they were largely dissatisfied. They were accustomed to C, had learned to deal with its complexities over assembly language, and had learned to accept its overheads. They were nevertheless surprised and amazed at the added complexities and overheads of C++.

On top of everything else was the dialect problem. C++ has been changing rapidly and steadily for years. "Standardization," a process that normally stabilizes a programming language, began in 1989. But in the case of C++, the standardization process has been used as an opportunity to add significant new capabilities to the language before it freezes. Multiple inheritance, exceptions, templates, and a host of other major features have been added over the past seven years or so, far faster than compiler writers can keep up. As a consequence, every commercial C++ compiler implements a different collection of language features. Moving a C++ program between compilers has become *more* difficult, not less, as a result of standardization—at least during this transitional period before the draft C++ Standard finally settles down and everyone gets

caught up.

It gets worse, at least from the standpoint of embedded systems programming. Several of the new features add significant overheads to a C++ program, in terms of both code size and execution speed. In some cases, the overheads occur even if you don't explicitly make use of the new features. Customers aren't asking for these new features—at least, they haven't yet been educated *en masse* to want the features—and the chip vendors haven't gotten around to implementing all of them. Compiler development in the commercial world is driven by what customers are asking for this month. There's simply too many things to do to devote product development time in places that don't confer an immediate competitive advantage.

Thus, some of the chip vendors might love to have an excuse not to have to supply, say, exception handling in C++. Many of their customers would happily live without such a feature, at least for the more demanding embedded programming projects. And once the customers discover the overheads associated with exception handling, they may well want to avoid those overheads even if the compiler does support the feature. Equally, the chip vendors don't want to be stigmatized. If the guy down the block sells a compiler with some sexy feature, it's hard to make a case for not providing it as well. Put simply, the vendors face a common software dilemma: how do you make sure that what the customers think they want is also close to what they really need?

A popular solution to such a problem is to define a suitable subset. In this case, we're talking about a subset of the full language and library mandated by the draft C++ Standard. Include in the subset everything that meets the need of embedded systems programmers. Omit anything that arguably can be left out, either because it adds to overheads, or it's too new to be widely available, or for some other good and proper reason. Get a bunch of people to agree on the subset. If several vendors supply the same subset, customers can write C++ code that is both

Embedded C++

efficient and portable across multiple implementations. Gone is much of the stigma for having less than a full implementation. In its place is, in fact, the cachet of matching a useful standard, even if it's only a de facto industry standard.

Thus was born the idea of Embedded C++ (or EC++, for short) as a dialect aimed squarely at the needs of embedded systems programmers. The group that first came together that Friday night in Tokyo soon dubbed itself the Embedded C++ Technical Committee. Hiroshi Monden of NEC became chair of the committee, and

Dr. Kiichiro Tamaru of Toshiba became vice secretary. ADaC is the secretariat. We Americans are merely advisers to the committee, which is very much a Japanese undertaking. The kickoff meeting was conducted mostly in English, as a courtesy to us Americans, who are notoriously weak at discussing technical matters in Japanese. Subsequent meetings were held in Japanese, rather more efficiently for the attendees I might add, throughout much of 1996. The committee maintained a reflector, and did a heckuva lot of work between meetings, in the bargain.

READ ALL ABOUT IT

I've been writing about C++ and embedded systems for over five years now, both in *Embedded Systems Programming* and in our sister publication, *C/C++ Users Journal*. For additional detail and considerably more background on the development of Embedded C++, you might want to review some of these essays. Most are available on CD-ROMs from Miller Freeman.

- "Embedded Programming in C++," *ESP*, November 1992, p. 97—some early musings on the role of C++ in embedded systems programming
- "Controlling Hardware from C and C++," *ESP*, December 1992, p. 73—the basics of low-level I/O in C, with some particular warnings about C++ issues
- "An Embedded C++ Library," *ESP*, October 1993, p. 113—how to adapt a hosted C++ library for embedded applications
- "From Indexes to Iterators," *ESP*, July 1995, p. 125—evolution of the iterator concept as it now appears in the Standard Template Library (STL)
- "A Taxonomy of Iterators," *ESP*, August 1995, p. 101—description of the different categories of iterators used in STL
- "Subsetting," *ESP*, March 1996, p. 109—early warnings about the thinking behind some of the EC++ effort, not yet announced
- "Figuring the Cost," *ESP*, May 1996, p. 117—description of some of the hidden overheads when programming in C++
- "Gedankenware," *ESP*, June 1996, p. 101—what happens to performance when you standardize before you acquire real-world experience
- "Too Much of a Good Thing," *ESP*, July 1996, p. 93—ways to cut some of the worst excesses of the draft Standard C++ library
- "Embedded C++," *ESP*, November 1996, p. 125—first formal announcement of the EC++ effort
- "Debugging Iterators," *ESP*, February 1997, p. 92—ways to detect subtle programming errors when using STL
- "Templates in C++," *ESP*, October 1997, p. 169—overview of template technology in C++, and a few programming tricks
- "Developing the Standard C++ Library," *C/C++ Users Journal*, October 1993, p. 10—first of a long series on the draft Standard C++ library. Practically every monthly column from this one through the present is devoted to describing the draft Standard C++ library. The columns between November 1995 and June 1997, in particular, are devoted to the Standard Template Library
- "Embedded C++," *C/C++ Users Journal*, February 1997, p. 35—a progress report on the EC++ effort

By September 1996, the Embedded C++ Technical Committee had produced a draft specification for their subset. They even had a Web site up and running. (See www.caravan.net/ec2plus.) As I indicated earlier, I got to make the first public announcement at the Fall 1996 Embedded Systems Conference. The project was out from under wraps and into the marketplace.

THE EC++ LANGUAGE

The Embedded C++ specification is a proper subset of the full draft C++ Standard. Thus, the most economical way to describe it is often in terms of what it doesn't have. I could list all the rules for writing declarations, statements, and expressions, for instance. That might reassure people who love detail that all the things they value about C++ (and C) are still there. But perspective is often lost in details. Saying what has been left out of EC++ is more economical, and often more revealing. Rest assured that what is left is a fully functional subset.

Programming languages such as C and C++ make a fairly sharp distinction between "language" and "library." The compiler is responsible for recognizing the language and generating code that carries out its intent. The library that accompanies the compiler provides a grab bag of classes, functions, and objects likely to be of use in many programs. I begin here by discussing just the EC++ language.

- Multiple inheritance and virtual base classes are omitted. These closely related features add quite a bit of complexity to the language, for a relatively small increase in expressiveness. Worse, they add overheads, however slight, to all calls to virtual member functions. (Have no fear, virtual member functions are still present.) A compiler explicitly designed to recognize the EC++ dialect can eliminate these overheads, if it chooses
- Runtime type identification (RTTI) is omitted. An even more recent addition to C++ than multiple inheritance, RTTI also adds complexity

Embedded C++

for a relatively small payoff in expressiveness. Runtime overheads for RTTI tend to be small, but they are present. As with multiple inheritance, such overheads can generally be eliminated only if the compiler has an EC++ dialect switch

- Exception handling is omitted. Strong arguments can be advanced both for keeping and omitting this language feature. Code that throws and catches exceptions can impose useful structure on the generally messy business of handling unusual conditions. But the space and time overheads imposed on all programs can be considerable, as you will soon see. Programmers of embedded systems often prefer to turn off this machinery and use simpler, if less well structured, mechanisms instead
- Templates are omitted. Adding templates arguably transformed C++ into a qualitatively different language. I've worked with templates enough to know that they are indeed powerful and often well worth the added complexity. But they are still a work in progress. The first commercial compilers that completely implement templates are just becoming available. Many products face years of development before they fully catch up. Using templates can also lead to surprising overheads. Change the type of one argument in one call to a template function and you can unwittingly double the amount of code generated to implement that function. Useful as they are, templates are well worth avoiding in an embedded context
- Namespaces are omitted. They were added to C++ a few years ago as a kind of directory structure for the global namespace. A principal intended use for namespaces was to partition the Standard C++ library so that it could be selectively replaced. But to date, I haven't seen a design that achieves this goal. The practical effect of adding namespaces to C++ is to require programmers to add a line to practically every existing C++ program that

effectively turns namespaces off. And the subtle effects of namespaces on already complex name lookup rules are still being discovered by implementors. This is another good idea that may well be worth omitting from an embedded development environment

- New-style casts are omitted. This is a tougher call. The type cast operator has been in C almost since its inception in the early '70s. But many programmers will agree that it needs to be refined. It sometimes does too much, covering up programming errors in the process. Draft Standard C++ introduces four new type cast operators to replace the old one, which is retained for backward compatibility only. Of these, only the dynamic cast has added runtime overheads. I suspect the new-style casts were omitted primarily on the basis of newness

Luckily for me, I was just an adviser to this effort. I can admire the designers for making tough decisions. At the same time, I don't have to take the heat for any controversial choices. It's a nice position to be in, for a change.

THE EC++ LIBRARY

Like the language portion, the Embedded C++ library is often best described in terms of what's missing. Much of what has been omitted from the full draft Standard C++ library must go out of necessity. Thus, there is no need for the header `<typeinfo>`, which works hand in hand with the `typeid` operator to implement runtime type identification. Keeping the headers `<exception>` and `<stdexcept>` is also hard to justify (though a case can still be made) if the language doesn't support exception handling.

If the language doesn't support templates, then any library template classes or template functions perforce must disappear. In some cases, it makes sense to replace a template class with an ordinary class, or a template function with an ordinary function. But in many cases, the only sensible thing is

to just drop the feature from the library. Thus, it's hard to preserve the Standard Template Library when there are no templates. My experience so far is that this is the chunk most sorely missed by programmers contemplating the use of EC++. But more on that topic later.

The `iostreams` library, on the other hand, began life in pre-standards days as a collection of ordinary classes. The classes became "templated" a few years ago, as part of a general conversion of the library to make greater use of templates. You can now traffic in streams of "wide characters," in order to support large character sets. You can even presumably invent your own kinds of streams. None of this latitude makes much sense in an embedded environment, however, and the hidden overheads have proven to be enormous. So, EC++ provides classes with the traditional names `istream`, `ostream`, and so forth. These classes are not retro artifacts, however. The EC++ `iostreams` classes behave just like their counterparts in the draft Standard C++ library that are implemented as template specializations.

You'll find a similar treatment of the complex arithmetic classes. You can't write the names `complex<float>` and `complex<double>` in a language that doesn't support templates. Instead, EC++ provides the classes `float_complex` and `double_complex` with the same behavior. This is one of the few places where EC++ is arguably not a pure proper subset of the full language and library.

A little more surprising is what's left out of the Standard C library. The draft Standard C++ library specification "includes by reference" all of the Standard C library, and then some. It mandates all the added support for wide characters from Amendment 1 to the C Standard. And it overloads a bunch of C functions in the bargain, to be more in keeping with C++ usages. The EC++ library keeps the overloads, but it omits all that extra baggage for supporting wide characters. Japanese developers were the ones who championed this addition to C—you'd think they'd be eager to keep it all. But the embedded

Embedded C++

community has different needs. Language and library support for wide characters is a luxury in this context.

For a more positive statement of what's actually *in* the EC++ library, see the sidebar, "EC++ Library Contents."

PERFORMANCE

The proof of the pudding is in the eating, as the saying goes. A fundamental premise behind the

design of Embedded C++ is that a subset language can yield greater space and speed efficiencies than the full C++ language. I've probably emphasized too much the need to avoid new language features, out of kindness to implementors and programmers alike. That is a transient problem that should cure itself over the next few years, as the C++ Standard becomes official and stable, compilers catch up with it, and

programmers learn how to use the new stuff. But performance issues will continue to be important for embedded systems for many years to come.

I wish I could present here a comprehensive list of benchmark results. My company, Dinkumware, Ltd., has such a project under way, but it is still in the preliminary stages. What I can present are some anecdotal results that I believe to be representative of actual programming situations. They compare relative program sizes for a couple of small test programs compiled in different ways and linked with different libraries.

The two libraries are the Dinkum C++ Library and the Dinkum EC++ Library. The former is a complete implementation of the draft Standard C++ library, in all its gory details. (See Microsoft VC++ v. 5.0 for an earlier version of this library.) The evidence to date is that this is a particularly efficient implementation, at least compared to the current competition. The latter library is Dinkumware's implementation of the Embedded C++ library. We also think it's pretty efficient, but so far we have nothing to compare it to. (See www.dinkumware.com for more information.)

Green Hills Software is a leading vendor of compilers for the embedded marketplace. They license the Dinkumware libraries, in all flavors, for distribution with their compilers. The two test programs were compiled with their PowerPC compiler that is now shipping. They generally observe that program size and execution speed are directly proportional, at least for typical test programs. My experience is that modern microprocessors generally behave this way—execution time depends primarily on the number of bytes of code the processor has to read. Barring hot spots, this is proportional to the total program size. So I show here only relative program sizes.

A small program that makes simple use of library I/O tends to be dominated by the contribution from the library. The relative sizes of such a test program are:

EC++ LIBRARY CONTENTS

The Embedded C++ library is a subset of the C and C++ libraries mandated by the draft C++ Standard. Here is a brief synopsis of what's required, on a header-by-header basis.

- The header `<cctype>`: functions `is*`, `to*`
- The header `<cerrno>`: macros `EDOM`, `ERANGE`, `errno`
- The header `<cmath>`: macros `DBL_*`, `FLT_*`, `LDBL_*`
- The header `<climits>`: macros `CHAR_BIT`, `*_MIN`, `*_MAX`
- The header `<locale>`: type `lconv`; function `localeconv`
- The header `<cmath>`: macro `HUGE_VAL`; functions (overloaded on `float`, `double`, and `long double`) `abs`, `acos`, `asin`, `atan`, `atan2`, `ceil`, `cos`, `cosh`, `exp`, `fabs`, `floor`, `fmod`, `frexp`, `ldexp`, `log`, `log10`, `mod`, `pow`, `sin`, `sinh`, `sinh`, `sqrt`, `tan`, `tanh`
- The header `<setjmp>`: type `jmp_buf`; macro `setjmp`; function `longjmp`
- The header `<stdarg>`: type `va_list`; macros `va_arg`, `va_end`, `va_start`
- The header `<stddef>`: macro `offsetof`; types `ptrdiff_t`, `size_t`
- The header `<stdio>`: types `FILE`, `fpos_t`, `size_t`; macros `EOF`, `NULL`; functions `fclose`, `fflush`, `fgetc`, `fgetpos`, `fopen`, `fputc`, `fsetpos`, `getchar`, `gets`, `printf`, `putchar`, `puts`, `scanf`, `setvbuf`, `sprintf`, `sscanf`, `ungetc`, `vprintf`, `vsprintf`; objects `stdin`, `stdout`
- The header `<stdlib>`: types `div_t`, `ldiv_t`; macros `MB_CUR_MAX`, `RAND_MAX`; functions `abort`, `abs`, `atol`, `atof`, `atoi`, `bsearch`, `calloc`, `div`, `free`, `labs`, `ldiv`, `malloc`, `qsort`, `rand`, `realloc`, `srand`, `strtod`, `strtol`, `strtoul`
- The header `<string>`: type `size_t`; macro `NULL`; functions `memchr`, `memcmp`, `memcpy`, `memmove`, `memset`, `strcat`, `strchr`, `strcmp`, `strcpy`, `strcspn`, `strlen`, `strncat`, `strncmp`, `strncpy`, `strprbrk`, `strrchr`, `strspn`, `strstr`, `strtok`
- The header `<complex>`: types `double_complex`, `float_complex`; functions (overloaded on `float_complex` and `double_complex`) `abs`, `arg`, `conjg`, `cos`, `cosh`, `exp`, `imag`, `log`, `log10`, `norm`, `polar`, `pow`, `real`, `sin`, `sinh`, `sqrt`, `tan`, `tanh`
- The header `<fstream>`: types `filebuf`, `ifstream`, `ofstream`
- The header `<iomanip>`: manipulators `resetiosflags`, `setbase`, `setfill`, `setiosflags`, `setprecision`, `setw`
- The header `<ios>`: types `ios`, `ios_base`; manipulators `boolalpha`, `dec`, `fixed`, `hex`, `internal`, `left`, `noboolalpha`, `noshowbase`, `noshowpoint`, `noskipws`, `noupper`, `oct`, `right`, `scientific`, `showbase`, `showpoint`, `skipws`, `uppercase`
- The header `<iosfwd>`: forward references to `iostreams` classes
- The header `<iostream>`: objects `cin`, `cout`
- The header `<istream>`: type `istream`; manipulators `endl`, `ends`, `flush`
- The header `<new>`: types `new_handler`, `nothrow`, `nothrow_t`; functions `operator delete`, `operator delete[]`, `operator new`, `operator new[]`, `set_new_handler`
- The header `<ostream>`: type `ostream`; manipulator `ws`
- The header `<sstream>`: types `istringstream`, `ostringstream`, `stringbuf`
- The header `<streambuf>`: types `fpos`, `streambuf`, `streamoff`, `streamsize`
- The header `<string>`: type `string`; function `getline`

Embedded C++

- 3 — EC++, no exceptions
- 5 — EC++ with exceptions
- 6 — Full C++, no exceptions
- 8 — Full C++ with exceptions

Note first the dramatic savings that occur when the compiler doesn't have to worry about exception-handling code. Your mileage may vary with your compiler, but exceptions are known to exact a significant toll on code size and/or execution speed. Then note what the simpler library gives you. By leaving out unwanted machinery that cannot be avoided in the full C++ library, EC++ weighs in at about half the size. Combine the language and library savings, and you get a program only three-eighths the size of full C++.

The effects are more dramatic for a somewhat larger program that makes greater demands on the library I/O facilities. In this case, it is harder for the full C++ library to avoid loading great quantities of additional code. The relative sizes are:

- 3 — EC++, no exceptions
- 5 — EC++ with exceptions
- 30 — Full C++, no exceptions
- 45 — Full C++ with exceptions

Yes, a program can truly weigh in at only 1/15 the size of full C++. Remember, we now live in a world where the traditional trivial program that simply prints "hello, world" can balloon to a full megabyte on some desktop implementations of C++.

Typical savings may be less dramatic, but the evidence to date is clear enough. There are indeed real performance advantages to be had by programming with the EC++ subset language and library.

THE ETC++ DIALECT

I don't want to pretend that the story ends here. For all the obvious success of Embedded C++ to date, a few counter trends are inevitable. The EC++ Technical Committee has faced a steady stream of gripes because one or another valued feature has been omitted from the subset. The pressure to expand any specification can be

enormous, to the point that it inevitably compromises early attempts at economy and/or simplicity. (Look at Java v. 1.1.) The Japanese committee members like their subset, and have so far stuck by their guns, but they don't want to appear unresponsive. And they recognize that other communities of programmers may have a different mix of needs than their chip customers.

Nobody wants a proliferation of dialects. The whole idea behind the formation of the consortium was to avoid such mayhem in the Japanese embedded systems community. Nevertheless, one significant new dialect has already appeared. And it could become at least as popular as EC++.

Green Hills has a major customer who decided last year to adopt Embedded C++ as an internal standard for embedded projects. But they also wanted to use the Standard Template Library. STL is a powerful tool with the nice property that you don't pay for it if you don't use it. (By contrast, the templated iostreams in the full C++ library impose unavoidable overheads on many programs.) So they got just what they asked for. Dinkumware supplied the necessary hybrid library to Green Hills, who in turn supplied the customer.

We were all pleasantly surprised to discover that this was a happy combination indeed. The EC++ library is about one-fourth the size of the full library. STL is about the same. So the combined library is still only half the full library, but it offers most of the functionality that programmers seem to actually want today. Compilers have to be pretty up-to-date with the draft C++ Standard to support STL well, but that's already not uncommon. Edison Design Group (www.edg.com) supplies the C++ front end for many embedded C++ compilers, and it is at the leading edge. It even has dialect switches already installed for tailoring the recognized language to be EC++ or many other interesting variants.

Dinkumware calls this combination of EC++ and STL the Dinkum Abridged Library. Green Hills has settled on the name Embedded Template

C++ (ETC++) for the combined language and library. Whatever the name, the dialect seems to be an effective compromise between EC++ and full C++. It retains the performance efficiencies of EC++, and offers the benefits of STL. (Sorry if this sounds like a sales pitch, but I'm genuinely enthusiastic about this dialect.)

The last important consideration is political. Many people have labored many years to complete the draft C++ Standard. Not the least among them is Bjarne Stroustrup, the original author of C++. It's hard to imagine these folks would enjoy seeing their work sliced and diced by others even before the ink dries on the draft. Everything that went into C++ during standardization went in for a reason. What right does some other committee have to second guess all these difficult choices?

In point of fact, there have been a few grumblings. But Stroustrup himself took the trouble to learn more about the EC++ project and has been gracious enough to give it his conditional blessing. He freely recognizes the need for subsetting, to meet specific goals. He makes it clear that he never intended C++ to be so large or inefficient that it is unsuitable for programming embedded systems. C++ has a long tradition in that arena.

For embedded programming, Stroustrup favors the largest dialect of C++ that avoids undue overheads. Thus, he prefers to leave in any features that are "weightless"—that add no overheads in space or speed. Namespaces and new-style casts are two obvious candidates for readmission, along with one or two other small items I've glossed over. I think it's fair to say that ETC++ meets his criteria pretty closely.

EC++ and ETC++ could benefit from some refining. A few small changes can simplify the business of moving a program between these two dialects and full C++. But these dialects have already broadened the possibilities for programming embedded systems in C++. Given the broad range of applications that fall under that umbrella term, we need all the elasticity we can get. ■