Engineering Essentials

BENJAMIN M. BROSGOL, Ph.D. | Senior Technical Staff, AdaCore
www.adacore.com

Comparing Ada and C

Both languages approach the reliability versus efficiency tradeoff from different angles, but each has a place in embedded-systems programming.

This issue is focused on the Internet of Things and the security issues that arise when interconnecting billions of devices, ranging from coffee makers to power grids. This article looks at the subject from a specific and rather basic perspective: Which language(s) should you choose to develop the software, where "software" means both the embedded code that runs the Things and the system programs that manage the networks, etc.? Choice of language is important since it affects the system's reliability, security, and performance, as well as the ease or difficulty in adapting the software as requirements change.

More specifically, this article compares C and Ada, summarizing their strengths and weaknesses and suggesting when to use (or not use) each. These two languages are interesting to look at: C because it's often the default choice for real-time and systems programming, and Ada because it has a successful (but not as well known) record in these same areas.

C and Ada have gone through various updates since their inception. I'll use the most recent version of each—C 11[1] and Ada 2012[2]—as the basis for the comparison. These reflect how the languages are evolving to meet current and future technological trends and challenges, even though at present it's more typical to find earlier versions of the languages in use.

C

In any kind of assessment, it always helps to go back to first principles. What were the main design goals for each language? The introduction to the 1999 version of the C standard[3] distilled the "spirit" of C into a small set of objectives, which have guided and constrained both the original design and each revision:

- Trust the programmer.
- Don't prevent the programmer from doing what needs to be done.
- Keep the language small and simple.
- Provide only one way to do an operation.
- Make it fast, even if it's not guaranteed to be portable.

In keeping with these principles, C offers various data types and data-structuring facilities (arrays, structs, pointers, unions, enums) with straightforward and efficient implementation, conventional algorithmic features (statements, expressions, functions), and modest modularization mechanisms (header files with function prototypes, #include directive, preprocessor).

Standard header files support dynamic memory management (malloc, free), a minimal exception mechanism (setjmp, longjmp), string handling, numerics, input/output, internationalization/locales, operating-system interfacing, and other services. Standard (but optional) and C++ compatible support for concurrent programming, including features that help exploit multicore platforms, have been introduced in C11. It specifies an explicit memory model, and supplies low-level facilities for thread management and communication.

By intent, C has some significant omissions. It doesn't provide generic templates (which can be approximated in part by the preprocessor), programmer-defined operator/function overloading, or object orientation, and its encapsulation support ("information hiding") is rudimentary.

In short, C is very much a WYSISWYG ("What You See Is What You Get") language. When you write a C program, you have a good idea of what the resulting compiled code and data will look like. Thus, C becomes a typical choice for low-level software that needs to interact directly with the hardware. However, a simple WYSIWYG language has two major drawbacks:

- It doesn't easily scale up to very large systems.
- In its focus on efficiency, it can sacrifice checks that are useful or necessary for reliability, safety, or security.

To somewhat address the latter point, "safe" subsets of C have been proposed over the years. Perhaps the best-known is MISRA C,[4] originally intended for automotive software but applicable to other domains as well. Static-analysis tools such as lint and a variety of commercial products have been used to detect potential vulnerabilities, although the language's weak type checking makes this more difficult than for other languages. And various guidelines have been published to facilitate secure coding.[5]

C11 has attempted to address some of the security issues via language features and libraries. For example, the optional Annex K (Bounds-checking interfaces) provides alternative versions of various standard functions, thus helping to prevent certain forms of buffer overflow as well as other vulnerabilities. The optional Annex L (Analyzability) constrains some forms of undefined behavior to be bounded, with the requirement that the implementation not perform an out-of-bounds store.

Will these new features be widely implemented, and will programmers use them? Time will tell. But in my opinion, they look like a patch that may mitigate some vulnerabilities but doesn't alter the original language philosophy. C wasn't designed for programming large-scale high-integrity applications. It's often selected not based on fitness for purpose, but because programmers know it (or it fits smoothly into an organization's software-development infrastructure), or because of perceived inefficiencies in other technologies.

ADA

Ada is very much at the other end of the spectrum. Perhaps a variation of C's principles serves as a first approximation to the "spirit" of Ada:

- Trust the programmer, but verify through appropriate checking since programmers are human and make mistakes.
- Prevent the programmer from doing what shouldn't be done.
- Keep the language kernel small and simple, but provide extension mechanisms in order to increase expressiveness.
- Provide one principal and intuitive

AN ADA PACKAGE

THIS CODELIST ILLUSTRATES a simple Ada package. The package specification, on the top, defines the Vector type as an array of Integer values. Different objects of this type can have different bounds. The Max function returns the maximum value in its parameter V. Its precondition is that V contains at least one element. Its post-condition captures the function's required semantics—the returned value has to be at least as large as every element in V, and it must be an element of V. The Negate procedure performs the unary "-" operation on each element in its parameter V. Its pre-condition (tao avoid overflow) is that no element can be the smallest Integer value. Its post-condition captures the procedure's semantics; V'Old is the value of V at the point of call. The contracts shown are appropriate for use with formal methods, so that they're verified statically, or they could be enabled as run-time checks to support debugging.

The package body contains the implementation of the two subprograms. V'First is the index of the lower bound of V, and V'Last is the index of the upper bound. The "for" loop in Negate illustrates the ability to iterate over a collection (here an array) without explicitly indexing. Note that Ada uses ":=" for assignment, "=" for equality, and "/=" for inequality.

```
package Math_Utilities is
  type Vector is array(Positive range <>) of Integer;

  function Max(V : Vector) return Integer
  with
    Pre  => V'Length>0,  -- V cannot be empty
    Post => (for all  Element of V => Max'Result >= Element) and
            (for some Element of V => Max'Result = Element);

  procedure Negate(V : in out Vector)
  with
    Pre =>  (for all Element of V => Element /= Integer'First),
    Post => (for all I in V'First .. V'Last => V(I) = -V'Old(I));
end Math_Utilities;
```

```
package body Math_Utilities is
  function Max(V : Vector) return Integer is
    Current_Max : Integer := V(V'First);
  begin
    for I in V'First+1 .. V'Last loop
      if V(I) > Current_Max then
         Current_Max := V(I);
      end if;
    end loop;
    return Current_Max;
  end Max;

  procedure Negate(V : in out Vector) is
  begin
    for Element of V loop
       Element := -Element;
    end loop;
  end Negate;
end Math_Utilities;
```

way to do an operation.

- Make it reliable and portable, and depend on the compiler to produce efficient code.

More generally, Ada's main goals were succinctly specified in the introduction to the first version of the language standard:

"Ada was designed with three overriding concerns: program reliability and maintenance, programming as a human activity, and efficiency."

More specifically, Ada was designed from the outset to take advantage of the breakthroughs in software engineering and programming methodology that occurred in the 1970s, with a focus on support for embedded real-time applications. The emphasis was on achieving confidence in program reliability (correctness), through features that include checks either statically or at run time.

Ada is a strongly typed extensible language, with facilities to define new types in various categories: integers, floating point, fixed point, enumeration, arrays, records (structs), and access types (pointers). Unlike C, Ada allows the definition of constrained subranges of scalar values, and checks ensure that objects aren't assigned out-of-range values. Subrange information is very useful to human readers and static-analysis tools.

Ada includes traditional algorithmic features, with a simple set of statements and with code modularization through subprograms (functions). It also has facilities for "programming in the large": encapsulation/data abstraction, separate compilation, packages (somewhat analogous to C header and code files), subprogram and operator overloading, generic templates, and full support for object-oriented programming (OOP). Ada also includes built-in features for exception handling and concurrency, including a structured feature for state-based mutual exclusion that helps avoid race conditions.

The predefined environment of Ada includes packages for character and string handling, I/O, numerics, containers, and operating-system interfaces. Ada also defines an annex with standard support for interfacing with other languages (including C), and optional specialized-needs annexes covering systems programming; real-time, distributed, and information systems; numerics; and safety and security (high-integrity systems).

Ada 2012 introduced contract-based programming features (pre- and post-conditions for subprograms, invariants for encapsulated types). This significant enhancement in effect embeds low-level requirements into the source code, with checks performed either at run time or (with appropriate tool support) statically. The Ada example *(see "An Ada Package")* illustrates the use of pre- and post-conditions; an analog example in C is shown in "C Header and Code File." Ada 2012 also increased the language's multiprocessor/multicore support and added a number of other enhancements.

Ada was intended for embedded systems, and programming at that level may involve getting down-and-dirty with the hardware—writing interrupt service routines, dealing with machine addresses and data representations, handling endianness issues, etc. With Ada, programmers can do all those things—one goal of the Systems Programming Annex is to give the programmer the tools to do anything in Ada that's possible in assembly language.

All of this sounds like a large and complex language. Indeed, the inclusion of generics, OOP, and exceptions makes Ada quite a bit more sizable than C, although subtleties in features such as sequence points don't make C the simple language as is commonly advertised. Skeptics might jest that, while a C program is WYSIWYG, Ada code seems more in the WTF category (acronym intentionally left unexpanded).

C HEADER AND CODE FILES

THE C HEADER and code files correspond to the Ada package *(see "An Ada Package")*. The pre-condition for max is modeled by an assert statement in the function body. The other Ada contracts are omitted, since C doesn't have quantification expressions.

One of the semantic differences between Ada and C concerns the treatment of array bounds. In Ada, the bounds are accessible through the array object via V'First and V'Last, while in C, the array size needs to be supplied as an explicit parameter to the functions.

```
// math_utilities.h

typedef int vector[];

int max(vector v, int n);
// n is the number of elements in v

void negate(vector v, int n);
// n is the number of elements in v
```

```
#include <assert.h>
#include "math_utilities.h"

int max(vector v, int n)
{
    assert(n>0);
    int curmax = v[0];
    for (int i=1; i<n; i++){
        if (curmax < v[i]){
           curmax = v[i];
        }
    }
    return curmax;
}

void negate(vector v, int n)
{
    for (int i=0; i<n; i++){
        v[i] = -v[i];
    }
}
```

QUESTIONS SURROUNDING ADA

Doesn't Ada have some performance challenges? And if Ada is supposed to be used for safety-critical or high-security systems, doesn't the semantic complexity get in the way? How do you certify a system where you need to show traceability from requirements down to object code, or where the implementation's run-time libraries are subject to the same certification requirements as the application software?

These are fair questions. Ada, like any other general-purpose language intended for high-integrity systems, needs to be constrained to a safe subset, only including features with well-defined behavior and a simple (certifiable) implementation. Ada actually anticipated this issue and supplies a compiler directive (pragma Restrictions) that allows programmers to specify features that will not be used. If using such a feature, then the error is detected, generally at compile time but sometimes at run time.

The Ravenscar tasking profile,[6] a set of Ada concurrency features with a small footprint and simple implementation, is part of the Ada standard and is defined through pragma Restrictions. Implementations can supply one or more restricted run-time profiles, corresponding to subsets at different levels of generality (and thus different levels of effort needed for certification).

Another notable example of an Ada subset is the SPARK language.[7] SPARK 2014, an Ada 2012 subset, is designed to facilitate formal proofs of program properties ranging from absence of run-time errors to compliance with a formally specified set of requirements. SPARK eliminates features that are hard to verify, such as pointers, but includes most of Ada's static semantics. Projects like the NSA-sponsored Tokeneer effort[8] demonstrated that ultra-high reliability and security is achievable with formal methods using conventional verification techniques.

CONCLUSIONS

C's emphasis has always been on performance, and its benefits show up most clearly when this requirement is critical (for example, in a software product for a competitive commercial market, where a customer's purchase decision may be strongly influenced by benchmarks). When reliability, safety, and/or security are overriding requirements, C has well-known defects.

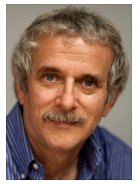
Historically, many security holes have been caused by writing past the end of an array, a bug that's detected in Ada. Some can be overcome with external tools (to enforce a "safe" subset or to detect vulnerabilities), or with the help of the new C11 features. However, the language wasn't designed with support for high-assurance systems as a major goal.

Ada's emphasis has always been on the various "ilities" (reliability, readability, maintainability), and its benefits show up most clearly when these requirements are critical (for example in a large, long-lived system where total software lifecycle costs need to be taken into account). Indeed, Ada (and safe subsets such as SPARK) has a long and successful usage history in safety-critical and high-security applications.

So when should Ada not be used? One context is when the need arises for rapid prototyping or scripting. Consider, instead, a dynamically typed language such as Python. Another scenario is when quickness to market is an important goal; then a higher software defect rate may be an acceptable price to pay.

How about when run-time performance (time, space) means the difference between a successful product and an also-ran? It's certainly possible to obtain efficient code from Ada, and indeed technologies such as gcc,[9] which incorporate a common code generator for multiple languages, yields the same performance for Ada and C on language constructs that have the same semantics. You can also improve efficiency by avoiding complex features, or by suppressing run-time checks after verifying through static analysis or sufficient testing that the checks will not fail.

Note that Ada versus C is not an "either/or" decision. They actually get along well together, largely due to Ada's standard interfacing support. An Ada program can import functions or global data from C, lay out data structures to have the same representation as the corresponding C data, and export subprograms or global data for use by an external C function. Therefore, a C program can be extended with functionality provided by Ada, and symmetrically, an Ada program can invoke C services.



DR. BENJAMIN BROSGOL, a senior member of the technical staff of AdaCore, has been involved with programming language design and implementation for more than 30 years. He was a Distinguished Reviewer of the original Ada language specification and a member of the design team for the Ada 95 revision.

REFERENCES:

1. ISO/IEC 9899:2011. Information technology -- Programming languages -- C.
2. Ada Reference Manual; ISO/IEC 8652:2012(E); Language and Standard Libraries. Available from www.adaic.org/ada-resources/standards/ada12/.
3. The C Standard Incorporating Technical Corrigendum 1; John Wiley & Sons; 1999.
4. MISRA C:2012 – Guidelines for the Use of the C Language in Critical Systems; www.misra-c.com.
5. Robert C. Seacord, Secure Coding in C and C++, 2nd Edition; Addison Wesley; 2013.
6. ISO/IEC TR 24718:2004. Guide for the use of the Ada Ravenscar profile in high integrity systems (2004).
7. SPARK 2014; www.spark-2014.org/.
8. Tokeneer ID Station Public Release Archive. Available from www.adacore.com/sparkpro/tokeneer.
9. GCC, the GNU Compiler Collection; gcc.gnu.org.