By Christopher Smith,
Director of Marketing, European Operations,
Green Hills Software.

Is C++ or Embedded C++ suited to your Embedded Application?

This article presents an overview of the capabilities of Embedded C++ (EC++). Embedded system software developers currently using C are faced with an important decision when beginning a new project - should they change their programming language to the increasingly popular C++, Embedded C++ (EC++) or remain with ANSI C? This article considers the benefits of using Embedded C++ versus C++ and the functionality it provides.

INTRODUCTION

Embedded system software developers today may desire to adopt the C++ programming language to benefit from features such as classes, templates, exception handling and class inheritance which have proven invaluable for native application development on desktop computers. However, most embedded system applications do not have tolerance to deal with the overhead and complexity of using C++. Some C++ features can dramatically increase the size of the application object code requiring additional target resources and impacting execution speed.

The object-oriented features of C++ can generally simplify the source code and therefore the development process, both by allowing code reuse and by placing the onerous housekeeping functions such as range checking and memory allocation in the class definitions and separate from the main application. Typically, C++ is more readable than standard C but once compiled, the C++ code size may be a factor of five larger than a C implementation.

EMBEDDED C++ IS A TRUE SUBSET OF C++

A true subset of ISO/ANSI C++, called Embedded C++ (EC++), has been developed by an industry group led largely by major Japanese CPU manufacturers. This group, known as the EC++ Technical Committee [1] set out to retain the object-oriented concepts of C++ whilst eliminating those most responsible for boosting memory requirements and reducing efficiency. This led to the development of the first EC++ compiler by Green Hills Software [2].

EC++ omits several C++ features such as multiple inheritance, virtual base classes, templates, exceptions, runtime type identification, virtual function tables and mutable specifiers. Whilst each of these features is useful in its own right, none is compelling for a sufficiently broad range of embedded applications. The support for some of these features will bloat the generated code, whether or not the features are actually used in an application. For example, exception han-

dling is one of the worst offenders and can adversely affect the deterministic response to external events required in real-time systems. So eliminating these particular features yields substantial reductions in the size of the compiler code and corresponding improvements in runtime efficiency.

DIFFERENTIATING C, EC++ AND C++

Through a series of examples, from a baseline EC++ implementation through to a full C++ implementation, the features that differentiate C, EC++ and C++ can be illustrated. The code fragment in Example 1 illustrates some the of key advantages that EC++ offers over the C language. The concept of classes is probably the single most important concept originally in C++ and also supported in EC++.

Classes build on the data structures found in the standard C language. In addition to allocating memory for a number of variables of mixed types, classes can be used to initialise variables, dynamically allocate additional memory for variables and arrays, perform range checking and many other useful functions. In C programs, these tasks have typically been distributed throughout the main code.

CLASSES AND OBJECT DEFINITION

In Example 1 is an illustration in the use of classes for an array operation. An embedded application such as a data-acquisition system might use such a class to create arrays for storage of data samples. The array class named Array includes two integer members. The first is a pointer named "elements" to members of an array and the second is used to track the array size.

As the main() portion of the code fragment implies, the declaration, Array a(6), can be used to create an array object with the name "a" that contains 6 elements. The class definition includes several important features such as the constructor code necessary to create the array and to ensure that the size specified is greater than zero. The constructor is located in the public section of the class definition which allows code located outside the class definition a window through which it

AD GREEN HILLS

C++

can access the elements and element_cnt class members. Each time an array of type Array is created, the compiler automatically calls the constructor function Array (int n). This function first assigns the value passed in the array declaration to element_cnt, then checks for a valid size, and finally allocates space in main memory for the array by calling "new".

In this simple example, a bad array size such as zero or a negative number causes the constructor to call a simple "die" function that outputs an error message. An embedded system would typically use a more elabo-

rate scheme to handle runtime errors.

The Array class also demonstrates two other key features of classes in EC++ or C++ -- function definitions within a class and overloaded operators. First consider the "size()" function which illustrates the simpler of the two concepts. Because element_cnt is a private member of the class, code outside the class can't directly access the counter. The size() function, however, allows the two "for" loops located in the main() section to indirectly access element_cnt for use as an upper limit of the loop.

EXAMPLE 1

```
//
// Embedded C++ example of a simple array class that does range checking
// on creation of an array, and on array subscripting operations.
//

#include <iostream>
extern "C" void exit(int);

// Deal with a runtime error.  A real embedded application would
// probably choose a different error handling strategy.

void die(const char *msg, int n)
{
    cout << msg << n << endl;
    exit(1);
}

// The integer array class

class Array {
private:
    int *elements;                      // array elements
    int element_cnt;                    // array size
public:
    Array(int n) : element_cnt(n) {     // construct a new array
      if (n > 0)
          elements = new int[element_cnt];
      else
          die("Bad Array size ", element_cnt);
    }
    int &operator [](int indx) const { // overloaded subscripting operator
      if (indx < 0 || indx >= element_cnt)
          die("Bad Array index ", indx);
      return elements[indx];
    }
    int size() { return element_cnt; } // return the size of the array
};
main()
{
    Array a(6);
    for (int i=0; i<a.size(); i++)
      a[i] = i;
    for (int i=0; i<a.size()+1; i++)   // error on a[7]
      cout << i << ". " << a[i] << endl;
}
```

Example 1.

OVERLOADED OPERATORS

The class definition also includes an example of overloaded operators. Operator overloading allows the programmer to develop new definitions of standard C/C++ operators such as "=", "›", or "+" that are customised for the type of object defined in a class. For
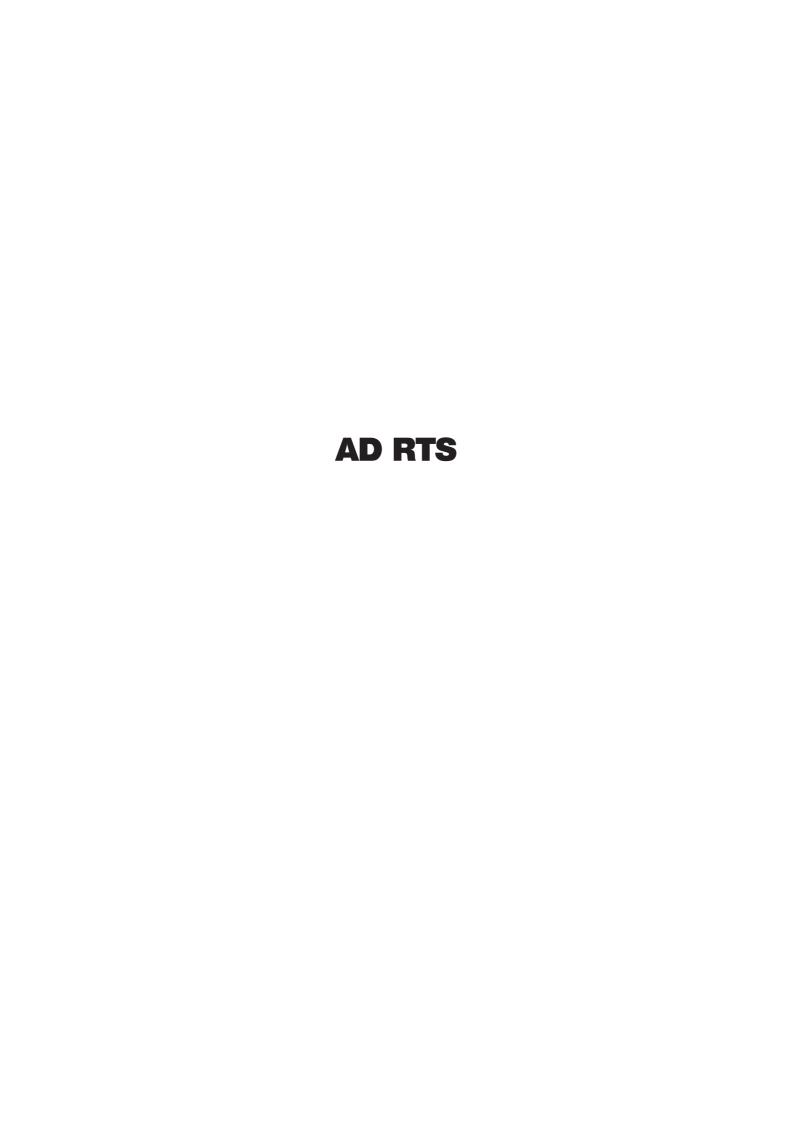
EXAMPLE 2

```
//
// Extended embedded C++ example of a simple array class that does range
// checking on creation on an array, and on array subscripting operations.
//
// This time we use a template class for the array class.
//

#include <iostream>
extern void die(const char *, int);
extern "C" void exit(int);

// Deal with a runtime error.  A real embedded application would
// probably choose a different error handling strategy.

void die(const char *msg, int n)
{
    cout << msg << n << endl;
    exit(1);
}


// The array class using templates

template <class T>
class Array {
private:
    T *elements;                        // array elements
    int element_cnt;                    // array size
public:
    Array(int n) : element_cnt(n) {     // construct a new array
      if (n > 0)
          elements = new T[element_cnt];
      else
          die("Bad Array size ", element_cnt);
    }
    T &operator [](int indx) const {    // overloaded subscripting operator
      if (indx < 0 || indx >= element_cnt)
          die("Bad Array index ", indx);
      return elements[indx];
    }
    int size() { return element_cnt; } // return the size of the array
};
main()
{
    Array<int> a1(6);
    for (int i=0; i<a1.size(); i++)
      a1[i] = i;
    for (int i=0; i<a1.size()+1; i++)  // error on a[7]
      cout << i << ". " << a1[i] << endl;

    Array<short> a2(6);
    for (int i=0; i<a2.size(); i++)
      a2[i] = i;
    for (int i=0; i<a2.size(); i++)
      cout << i << ". " << a2[i] << endl;
}
```

Example 2.

C++

AD RTS

example, a class could be developed that defined an object such as a circle or sphere. A size comparison could be made of two objects that were created using the class definition -- objects A and B. The expression A›B or A=B have well understood meanings when A and B are integers, but the compiler could have trouble evaluating the expressions when A and B are spheres of given size or composition. To eliminate ambiguity, EC++ and C++ permit the programmer to include a new definition for such operators that works in a way that is advantageous to the specific application and object type.

The sample code in Example 1 overloads the subscripting operator "[ ]" used to store the index for an array. In this case, the overloaded function gets called each time an indexed array reference occurs -- for example "a[i]=i. Instead of changing the effective meaning of the subscripting operator, the example uses the overloaded operator to automatically detect for out-of-range array indices. Note that should the sample program be executed, the output statement used in the second loop would generate an error on the sixth pass through the loop because a[7] would exceed the valid index test.

It can be seen that classes significantly streamline the mainline code in an EC++ or C++ program. For example, a C program would require explicit data-structure definitions for every array declared while EC++ or C++ handles creation of all similar objects with a single class. Moreover, C programs would require memory-allocation, error-checking and element-count code in the main part of the program or in dedicated C functions. The compiled code overhead of EC++ relative to standard C code is minimal as well. Adding classes and overloaded operators only marginally increases the generated code size.

Important Note: These code savings cannot be realised simply by not using the memory-hungry C++ features in an application and then compiling the code with a standard C++ compiler. See the highlighted section "Why Use An EC++ Compiler?" details the consequences of attempting this approach.

USING C++ FUNCTIONS OMITTED FROM EC++

Having looked at the advantages offered by EC++, some omitted C++ functions prove to be extremely desirable for given applications. To this end, compiler vendors can provide programmers with some flexibility as to what C++ functions are available for use in each application. For example, a programmer can use a compiler directive with the Green Hills C++/EC++ compiler to strictly limit the source code to the EC++ subset and fully realise the savings in code size and the boost in efficiency. Additionally, a programmer can use compiler switches to add support for one or a few specific C++ functions that were left out of EC++. The granular support for optional C++ features allows the programmer to trade off compiled code size with ease of development and maintainability. Furthermore, the Green Hills C++/EC++ compiler allows the programmer to choose libraries appropriate to their application thereby eliminating a significant amount of redundant library code.

ADDING TEMPLATE SUPPORT

Templates provide an excellent example of a C++ feature not included in EC++ that provides significant advantages in development with only a modest increase in memory requirements when used carefully. Example 2 illustrates the benefit of templates. In Example 1, the Array class was defined in such a way that all members of the array had to be integers. With EC++, additional classes would have to be defined to handle arrays for short, long, char or floating-point data

WHY USE AN EC++ COMPILER?

Since EC++ is a proper subset of ISO/ANSI C++, one might assume that the efficiencies promised by EC++ could be delivered by simply avoiding certain C++ features. EC++ code can certainly be compiled on a C++ compiler but the resulting code will still require 5 or 6 times more memory than the same code compiled with an EC++ compiler.

There are three main problems that arise from attempting to increase efficiency using EC++ code and a standard C++ compiler.

First problem: The compiler will still use C++ libraries and link a multitude of code into the finished product that is surplus to requirements for your application. An EC++ compiler, conversely, uses libraries that are optimised for the new dialect and thereby generates smaller more efficient code.

Second problem: The compiler is not as efficient in code optimisation. EC++ compilers can achieve superior optimisation relative to C++ compilers. EC++ compilers can optimise code without presuming the possibility that complex features such as exception handling may be used at some point.

Third problem: Standard C++ compilers have no mechanism to enforce EC++ compliance within a programming team. With a standard C++ compiler, one out of 10 or 20 programmers on a team can use an offending feature and destroy the efficiency of the entire code base.

Green Hills developed a sample EC++ program to demonstrate memory efficiency. The program solves a form of the classical "travelling salesman" [3] problem often used to teach programming. When compiled on the Green Hills C++/EC++ compiler using EC++ mode the total code size is 57 Kbytes. When compiled using the identical EC++ source file but in C++ mode with no exception handling library, the code size is 322 Kbytes. Adding an exception-handling library brings the code size to 378 Kbytes.

types. Templates allow a single class definition to support creation of arrays for any valid C++ data type.

The only real addition to the Array class definition with a template is the template label that precedes the class definition and the use of the "T" specifier each time the code addresses an element of the array. Consider the main() code, however, and see that Array works equally well to instantiate array "a1" to store integer data types and array "a2" to store short data types. It would be just as easy to define more arrays to store other data types. An embedded system performing data acquisition, for example, might well require floating-point arrays.

The use of templates as illustrated would result in little or no increase in compiled code size, so programmers that don't need to strictly meet the EC++ specification can leverage a valuable tool. Caution: the code size realised will depend specifically on the embedded application. Much of the code bloat found in C++ code comes not from using a feature such as a template but from referencing templates that are found in large C++ libraries. Reference one of these standard templates, and the resultant code will include many aspects of the library that are not required.

Experienced C++ programmers will enable a number of C++ features in an EC++ environment and not decrease efficiency. Such an environment might be called extended EC++. The EC++ development effort eliminated features such as templates, namespaces, mutable specifiers, and new-style casts more so due to the complexity of using the features properly than due to inherent inefficiency. C programmers may find it difficult to use the features correctly, but careful, experienced C++ programmers can leverage the benefits of these features without penalty. Moreover, compilers such as the Green Hills C++/EC++ compiler makes extending EC++ as simple as using compiler switches.

FULL C++ WITH EXCEPTION HANDLING

What other features become available in moving towards full C++? Exception handling proves to be among the most valuable of features to embedded system designers, yet is also among the leading in causing compiled code bloat. Exception handling provides a systematic approach to trapping errors caused by operator input or even out-of-range errors in a data-acquisition environment.

The code fragment in Example 3 illustrates C++ exception handling. Actually, the example is more typical of the kind of error handling required in complex embedded systems than was the simple die() function used in the first two examples.

C++ defines the keywords "try", "throw", and "catch" for use in exception handling. Typically, programmers organise code within blocks called try blocks that are enclosed within braces -- {}. A second block of code called the catch block is dedicated as a centralised runtime exception/error dispatch service. Anywhere within the try block, a throw directive can originate an exception condition and transfer control to the catch block based on evaluation of a C++ "if" statement. The throw statements can be in-line within the try block or located in functions within the class definition.

Example 3 uses the throw mechanism at two different places in the Array class definition. The first throws to the catch block when an Array instantiation has zero or a negative number of elements. This throw is located in the constructor function. The second throw is used to handle array index errors detected by the overloaded array subscripting operator.

C++ offers significant flexibility in how exceptions are handled and in all cases allows separation of the exception handling code from the mainline application. As in the example, it is possible to dedicate a catch block to each try block. Alternatively, define a single catch block to service an entire main() program. The code in a catch block only executes when a throw evaluation fails. For example, should the code within the first try block in Example 3 execute with no exception the following catch block will be skipped and the second try block will execute.

FINDING THE RIGHT MIX FOR A SPECIFIC APPLICATION

EC++ promises to provide embedded-system programmers a valuable path to leverage the most significant aspects of an object-oriented language. With formal approval of the EC++ standard imminent, programmers should demand C++ compilers that include EC++ support. To minimise development time and simplify code maintenance, however, programmers should not dismiss all of the C++ features that were eliminated in EC++. By carefully choosing which features to use on an application-by-application basis, programmers can both simplify development and maintain reasonable runtime efficiency. Moreover, complex embedded systems easily benefit from the robust libraries and features of C++. The Green Hills C++/EC++ compiler offers full support for the EC++ specification, and the flexibility to re-activate individual features of the C++ language. Moreover, programmers can select from a variety of libraries to match the feature set being used. ■

Christopher Smith is Director of Marketing, European Operations for Green Hills Software, Inc. and is located in the UK. He has over fifteen years of technical sales, marketing and international distribution experience in the embedded systems marketplace with compilers, debuggers, real-time operating systems, in-circuit emulators, logic analysers and CAE products from world-leading suppliers. An electronics design engineer by profession, Chris left his engineering position at Penny & Giles to follow a career in technical sales with ARS Microsystems Ltd where he remained for nine years. In 1994, he left his Sales Directorship to start-up the European Operation for Intermetrics Microsystems Software, Inc. (later acquired by TASKING, Inc.), developer of the InterTools range of compilers and debuggers. In September 1998 he joined Green Hills Software, Inc. with responsibilities for pan-European marketing activities.