

Embedded C++ decreases code size, enhances execution speed

Exploiting the object-oriented features of C++, EC++ provides an embedded-software-development platform that is streamlined and scalable.

BY MIKE HADEN
Green Hills Software

C++ is one of the most sophisticated programming languages around. Programmers like it because it is more powerful than standard C code, providing object-oriented features (classes, templates, exception handling, and class inheritance) that greatly simplify the development process. Despite its many favorable features, however, C++ has a downside. Once compiled, C++ code can swell by a factor of five or more relative to a standard C implementation, significantly increasing memory requirements and reducing execution speed on target systems.

One way to reap the development advantages of C++ without incurring the code bloat is to use a subset of C++ known as EC++. For the purposes of most embedded-development projects, EC++ retains the most valuable features of C++ while discarding those responsible for code bloat, including multiple inheritance, virtual base classes, templates, exceptions, run-time-type identification, virtual function tables, and mutable specifiers. While each of these features is helpful in its own right, none is sufficiently compelling for a broad range of embedded applications. Eliminating these features from the compiler can yield signifi-

cant reductions in code size and significant improvements in run-time efficiency, even if they are not used in the application.

WHY C++?

The creation of classes, supported in EC++, is probably the single most im-

portant concept originally brought about by C++. 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 initialize variables, allocate additional memory for variables and arrays, perform

Example 1

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

#include <iostream>

extern "C" void exit(int);

// Deal with a runtime error. A real embedded application would
// most likely 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
};

int 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;
    return 0;
}
```

range checking, and perform many other useful tasks. In C programs, these tasks have typically been scattered throughout the main code.

Example 1 illustrates the use of classes for an array operation. The class named Array includes two integer members. To track the array size, “element_cnt” is used; “elements” is a pointer to other members of an array.

You can use the declaration, Array a(6), to create an array object with the name “a” that contains six elements. The class definition includes several important features, such as the constructor code necessary to create the array and 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 can access the elements and element_cnt class members. Each time an array of type Array is created, a 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 it “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, which outputs an error message. An embedded system would likely use a more elaborate scheme to handle run-time errors.

The Array class also demonstrates two other key features of classes in EC++ and 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.

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 “+,” which are customized for the type of object defined in a class. For example, you could develop a class that defined an object such as a circle

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
};

int 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;
    return 0;
}
```

or sphere. Next, you might want to compare the size of two objects that were created using the class definition—objects A and B. The meaning of the formulas $A > B$ and $A = B$ have well understood meanings when A and B are integers, but a compiler could have trouble evaluating the expressions when they are spheres of given size or composition. To eliminate ambiguity, EC++ and C++ allow the programmer to include

a new definition for such operators that are customized for 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.” Rather than changing the effective meaning of the subscripting oper-

Example 3

```
//
// 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 and C++
// exception handling to deal with runtime errors.
//

#include <iostream>
using namespace std;

// The array class using templates
template <class T>
class Array {
private:
    T *elements;           // array elements
    int element_cnt;       // array size
public:
    class Range {          // A nested class to deal with
public:                    // runtime errors
        int indx;
        char *msg;
        Range(char *m, int i) : msg(m), indx(i) {}
    };
    Array(int n) : element_cnt(n) { // construct a new array
        if (n > 0)
            elements = new T[element_cnt];
        else
            throw Range(" Bad Array size ", element_cnt); // runtime error
    }
    T &operator [ ](int indx) const {
        if (indx < 0 || indx >= element_cnt)
            throw Range(" Bad array index: ", indx); // runtime error
        return elements[indx];
    }
    int size() { return element_cnt; } // return the size of the array
};

int main()
{
    // arrange to catch runtime errors related to the Array<int> class
    try {
        Array<int> a1(6);
        for (int i=0; i<a1.size(); i++)
            a1[i] = i;
        for (int i=0; i<a1.size(); i++)
            cout << i << " . " << a1[i] << endl;
    }
    catch (Array<int>::Range rng) { // deal with runtime errors here
        cerr << rng.msg << rng.indx << endl;
    }

    // arrange to catch runtime errors related to the Array<short> class
    try {
        Array<short> a2(-1);
        for (int i=0; i<a2.size(); i++)
            a2[i] = i;
        for (int i=0; i<a2.size(); i++)
            cout << i << " . " << a2[i] << endl;
    }
    catch (Array<double>::Range rng) { // deal with runtime errors here
        cerr << rng.msg << rng.indx << endl;
    }
    return 0;
}
```

ator, the example uses the overloaded operator to automatically detect out-of-range array indices. Note that if you execute the sample program, 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.

As the example shows, classes significantly streamline the mainline code in an EC++ or C++ program. A C program would require explicit data-structure definitions for every array declared, while EC++ or C++ handle 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.

BRIDGING THE GAP

The baseline EC++ definition eliminates many C++ features such as templates, name spaces, mutable specifiers, and new-style casts, not because they are inherently inefficient, but because proper use of these features requires great experience and care. However, experienced C++ programmers can leverage the benefits of these features with no penalty. Furthermore, experienced C++ programmers can often add C++ features to the baseline EC++ environment without significantly decreasing efficiency.

Programmers can use a compiler switch to strictly limit the source code to the EC++ subset. They can also use compiler switches to add support for one or more C++ functions that were left out of EC++. This approach enables programmers to make tradeoffs between compiled code size, development ease, and maintainability that are best for their application. For example, programmers can use switches to enable a portion of the C++ libraries while eliminating significant amounts of unneeded library code.

ADDING TEMPLATES

Templates are an example of a C++ feature not included in EC++ that, when used with care, can be added without significantly increasing memory requirements. 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. EC++ requires you to define

additional classes even if you only want to handle arrays for short, long, char, or floating-point data types. Templates, on the other hand, allow a single class definition to support creation of arrays for any valid C++ data type.

The principal benefit that templates add to the Array class definition 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 you will see that Array works equally well to instantiate array "a1" to store integer data types and array "a2" to store short data types. You could just as easily define more arrays to store other data types. An embedded system performing data acquisition, for example, could very well require floating-point arrays.

The use of templates, as illustrated on page 35, would result in no increase in compiled code size versus an equivalent implementation without templates. Beware, however, that the code size you realize will depend specifically on your application. Much of the code bloat found in C++ code comes not from using features such as templates, but from referencing templates that are found in large C++ libraries. Reference one of these standard templates, and you end up compiling many things in the library that you don't need.

EXCEPTION HANDLING

Another useful feature of C++, particularly

to embedded designers, is exception handling, which provides a systematic approach to trapping operator input and out-of-range errors in a data-acquisition environment. Unfortunately, exception handling is also a leading cause of compiled-code bloat. Thus, it is not supported in the baseline EC++ definition and should be added with care.

The code fragment in Example 3 illustrates C++ exception handling. This 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 organize code within blocks called *try blocks* that are enclosed within brackets—{ }. A second block of code called the *catch block* is dedicated as a centralized run-time 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 incorporated inline 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 handles array index errors

detected by the overloaded array-subscripting operator.

C++ offers significant flexibility in how exceptions are handled and in all cases allows you to separate the exception-handling code from the mainline application. As in the example, you can dedicate a catch block to each try block. Alternatively, you can 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, if the code within the first try block in Example 3 executes with no exception, the following catch block will be skipped and the second try block will execute.

FINDING THE RIGHT MIX

EC++ provides a baseline platform for embedded software development that enables programmers to leverage the object-oriented features of C++ while minimizing code bloat. EC++ is also scalable, enabling programmers to add select C++ features not supported in the baseline definition to their applications, often with minimal impact on code size and run-time efficiency. With EC++ having the status of a formal standard, programmers should demand C++ compilers that include EC++ support. ♦

Mike Haden is director of engineering, advanced products for Green Hills Software (Santa Barbara, CA).

