The Inefficiency of C++, Fact or Fiction?

Anders Lundgren, IAR Systems
Anders.Lundgren@iar.se

ABSTRACT

A widespread "truth" among developers of embedded software is that using C++ results in inferior code size and speed compared with using C. This article will attempt to sort out the facts from the fiction in this statement. By better understanding the underlying mechanisms of the language, a designer can avoid code bloat.

In the article we will discuss various C++ language features, compare them with C, describe their implications for the ARM code generation, and look at the efficiency of the different ARM architectures.

INTRODUCTION

C++ offers the embedded programmer some striking advantages over C. As a starting point, it can be seen simply as a better C, some C programmers will run their code through C++ compilers for quality checking. This also demonstrates that moving to C++ is not an all or nothing event, it is possible to choose among the C++ features those that are useful in the application and ignore others, just as this article will do. The stronger type checking, compared with C, means that many errors are caught at compile time. It is possible to exercise tight control over memory locations and exploit the knowledge for tighter execution code. And the change of approach that comes with object orientation provides improved debugging and maintenance.

But alongside these recognized advantages, there seems to be a general feeling among C programmers that C++ can result in inefficient code when compared with coding the same application in C. Like all such general knowledge, this need not be true, it all depends on which C++ features you use and how you use them.

TARGET SYSTEM

For the purposes of this discussion, the target is a typical small embedded system where code size is a concern and where the flash available is between 64kByte and 512kByte.

EVALUATION OF C++ CONSTRUCTS

The C++ constructs that will be considered are

- encapsulation/classes
- namespaces
- inlining
- operator overloading
- constructors/destructors
- references
- virtual functions
- templates
- STL (Standard Template Library)
- RTTI
- exceptions

C++ constructs and the code will be given price tags, compared with C code achieving the same result. Something priced as FREE has no overhead compared to coding in C (not that no code is generated, just that the size of the generated code will be almost the same as in C.) CHEAP has a small overhead compared to C while EXPENSIVE has a large overhead compared to C. All the comparisons were made using the C/C++ ARM compiler from IAR Systems.

Encapsulation – information hiding

In C++, encapsulation is implemented by the `class`, which is the equivalent of a C `struct` but with the addition of methods (functions) and an information hiding mechanism; the private part is accessible only from within the class `CircularBuffer` while the public section is accessible to all users of the class.

```
class CircularBuffer
{
private: // implicit private
      unsigned char mBuffer[256];
      unsigned char mFirst;
      unsigned char mLast;
public:
      CircularBuffer() : mFirst(0), mLast(0) {} // constructor
      ~CircularBuffer() {} // destructor
      bool IsEmpty() { return mFirst == mLast; } // implicit inline
      bool IsFull();
      void Write(unsigned char c);
      unsigned char Read();
};
```

Example 1: Class

By looking at how the compiler treats the C++ code internally we will get a good understanding of the cost associated with using classes. Two examples will be used as illustration.

The first example (2) shows how the implementation of larger methods (functions), such as `Write`, are not done inline. The function names use the global name space and are mangled to form unique names, in a real implementation the parameter and return types are also part of the mangled function name. The example also shows how the pointer `this` is used as a hidden parameter from the C++ point of view.

```
void CircularBuffer::Write(unsigned char c)
{
      if (IsFull()) Error("Buffer full");
      mBuffer[mLast++] = c;
}
```

Example 2 (a): C++ class member function

```
void CircularBuffer_Write(struct CircularBuffer *this, unsigned
char c)
{
     if (CircularBuffer_IsFull(this)) Error("Buffer full");
     this->mBuffer[this->mLast++] = c;
}
```

Example 2 (b): How the function is handled internally by the compiler

In the next example (3), the object `buf` is created as an instance of the class `CircularBuffer`. A pointer to the object, `p`, is also created. A hidden parameter, called `this`, which is also a pointer to the object, is passed to each call of a member function. Although the syntax gives the impression that the functions are called through pointers, they are actually called directly as seen in (3b).

```
CircularBuffer buf;
CircularBuffer *p = &buf;

void test()
{
      buf.Write('a'); // Call member function directly
      p->Read(); // Call member function through pointer
}
```

Example 3 (a): C++ object creation and calling member functions

```
void test()
{
     CircularBuffer_Write(&buf, 'a');
     CircularBuffer_Read(p);
}
```

Example 3 (b): How the function is handled internally by the compiler

The function call is made in the same way as in C. Setting up a pointer to the object and passing it to the member function is likely to be the same in C as it is in C++. For comparison, then, the cost of function calls and use of pointers are FREE.

Namespace

Within a namespace, all visible names are grouped together. Code outside the namespace wanting to refer to data within the namespace, must qualify with the namespace name. In the example, this could be `Decoders::bitrate`. Namespace has no cost associated with it: FREE.

```
namespace Decoders
{
      int bitrate;
      ...
}
```

Example 4: A namespace declaration

Implicit inlining

Inlining, running a copy of a function rather than calling the function, initially sounds expensive, but, by avoiding the function call overhead which can be considerable for small member functions leads to a dramatic code size reduction. In C++, functions defined in the class are by default inlined. Inlining in C++ is essential for good code generation and the cost is therefore FREE.

```
class CircularBuffer
{
public:
      bool IsEmpty() { return mFirst == mLast; } // implicit inline
};
```

Example 5: Implicit inlining

Operator overloading

With operator overloading it is possible, within a class, to define the function of a standard operator (such as `+`, `-`, `|` …) to operate on the class. In the example, the + operator is defined not as a simple addition, instead it is used to concatenate two circular buffers. When the + operator is encountered, it is translated to a function call to the overloaded operator function.

```
CircularBuffer operator+(const CircularBuffer& a, const
CircularBuffer& b);

CircularBuffer buf, buf_a, buf_b;

buf = buf_a + buf_b;
```

Compiled code for the + operator

```
LDR R2,=buf_b
LDR R1,=buf_a
LDR R0,=buf ; return value
BL CircularBuffer_operator_plus
```

Example 6: operator overloading: + is defined as function within the class. Note that R0 points to the return value location.

If used in a natural way, such as using + for imaginary numbers and concatenation operations, operator overloading is a powerful way of simplifying code writing and it is FREE.

Constructor and Destructor

When an object is created, the constructor function is called implicitly. The created object can be, as in the example, member data, or the constructor can be used for hardware initialization, for example when creating an UART object. The destructor is also called implicitly when an object is destroyed; this provides better control of the memory space and makes maintenance easier.

```
class CircularBuffer
{
     public:
     // constructor
     CircularBuffer() :
     mFirst(0), mLast(0) {}
     // destructor
     ~CircularBuffer() {}
};
```

Example 7: constructor and destructor

Constructors and destructors are essentially FREE except for the actual code they contain.

References

References, which are mostly used as parameters, is a safer way to do call-by-reference compared to using pointers.

```
void get5(int& value)
{
      value = 5;
};

                  will be compiled to   MOV R1,#+5
                                        STR R1,[R0, #+0]
```

Example 8: References

References have the same cost as passing a pointer, and are FREE.

Inheritance and virtual functions

When different classes have data and functionality in common, the classes can share data and methods through inheritance. By combining inheritance with virtual functions it is possible to separate the class interface from the implementation, retaining clarity and making future changes easier.

The example shows how inheritance can be used in developing a portable music player, able to play MP3, WMA and Ogg formats.

The class `Track` represents the properties and functionality needed to play a track, specifying the interface between a track and the rest of the world. Other classes will inherit from `Track`. As it is an abstract class, no objects of the type `Track` can be created. A class is made abstract if at least one of the virtual functions are declared "pure virtual" using the notation `=0`, meaning that the function must be defined in the derived class.

```
class Track // base class
{
public:
      virtual string const& Artist() = 0;
      virtual string const& Title() = 0;
      virtual void Play() = 0;
};
```

Example 9 (a): Defining the abstract class, `Track`

The derived class, `MP3Track` inherits from the base class `Track`. The derived class must implement all the pure virtual functions in `Track`. The derived class `WMATrack` will look the same, but will decode WMA instead of MP3. The separation of interface and implementation means that in `DoMusic` there is no need to know what type of track is being decoded. `p->Play()` will call `Mp3Track::Play()` or `WmaTrack::Play()` depending on which track implementation `p` points to.

```
class Mp3Track : public Track // derived class
{
public:
      virtual string const & Artist(); // Extract Artist info from ID tag
      virtual string const & Title();  // Extract track title info from ID tag
      virtual void Play();             // Play the audio data
};

void DoMusic(Track *p)
{
     p->Play();
}
```

Example 9 (b): Creating the derived class, `MP3Track`

To implement this, each object gets a `vptr` that points to the `vtable` for the corresponding class, and the `vtable` has one function pointer for each virtual function.

```
void Mp3Track_Play(struct Mp3Track *this);
typedef void (*Fptr)(struct Mp3Track *);
typedef Fptr (VTable)[4];

struct Track
{
      VTable const *vptr;  // Hidden element
};
struct Mp3Track
{
      struct Track mBase;  // Hidden element
};

const VTable Mp3Track_vtable =
{
      (Fptr)Mp3Track_Artist,
      (Fptr)Mp3Track_Title,
      (Fptr)Mp3Track_Play
};

void DoMusic(Track *p)
{
      (*(p->mBase.vptr[2]))(p);
}
```

compiles to

```
LDR R1,[R0, #+0]
LDR R1,[R1, #+12]
MOV LR,PC
BX R1
```

Example 9 (c): How the virtual function mechanism is handled internally by the compiler, by implementing `vtable` and `vptr`

The player could be implemented in C, in many different ways. These would include using a `switch` statement on the track type, or through a table lookup and function call through a function pointer. All functions, `Artist`, `Title` and `Play` would need to be implemented using one of these mechanisms. So instead of one `vtable` for each class, there will be a switch table or a table of function pointers for each function. This would tend to spread the track details all over the code, making it a non-trivial operation to add support for a new track type, for example adding the AAC audio format. Debugging and maintenance will also be more complex.

Using virtual functions does come with a price. There needs to be one `vptr` per object, and one `vtable` per class for virtual functions as calls to virtual functions must follow `vptr` and lookup the function address in `vtable`. In C, by contrast, the function address can be looked up in a table without following a pointer. In C++, each created object needs 4 bytes, compared to a table in C. It also needs constructor code to set up `vptr` and `vtable` data.

However, the code price paid is still CHEAP, and for many cases can be considered as almost FREE. In addition, virtual functions provides huge advantages for future code maintenance and product development.

Templates

Templates, once called type parameterization, provide a way of using a piece of code in several different variants in an application, like a macro. This cuts down on code writing and simplifies maintenance; a change to the template is instantly reflected in every instantiation of that template. Once written, a template can be used in other applications; an example of re-usable code.

Templates come in two flavors: Function Templates and Class Templates. Function templates are closer to macros but are more secure both syntactically and semantically. As with macros, each invocation potentially generates extra code.

```
/* C implementation */
#define CastToInt(x) ((int) x)
int FloatToInt(float f)
{
      return CastToInt(f);
}

// C++ implementation
template<typename X> int Cast2Int(X x)
{
      return (int) x;
}

int Float2Int(float f)
{
       return Cast2Int(f); // Implicit instantiation.
}
```

Example 10 (a): Function template

Class templates offer a lot more as each template creates a class, with all the functionality of a class, including the ability to include templates.

Templates provide the opportunity to build elegant implementations of very complex structures for reuse, but, as always complexity comes with a potential for significant code size expense, at least for the template user if not the template implementer.

```
template<typename X> class Value {
public:
      Value(X x) : mX(x) {
      }
private:
      X mX;
      };

Value<int> val1(6);
```

Example 10 (b): Class template

But elegance does not have to be expensive, this example shows how a template can compute values at translation time, rather than at runtime, generating no code but only the constant 24.

```
template<int N> class Factorial {
      public:
             static const int value = N * Factorial<N-1>::value;
};

class Factorial<1> {
      public:
      static const int value = 1;
};

// factorial = 24 (1*2*3*4)
int factorial = Factorial<4>::value;
```

Example 10 (c): Example template: the template is recursively expanded for the values 4, 3, 2 and 1 and the factorial is calculated at translation time.

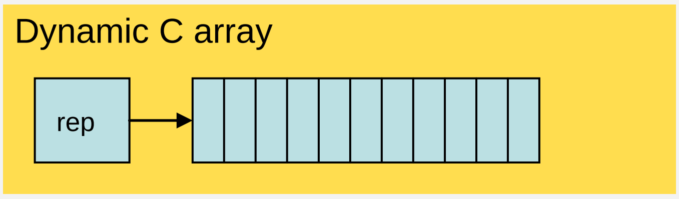
Because template complexity can vary enormously, from a simple macro expansion with a couple of lines of code to a complex set of functions generating lots of code, the cost also varies. Templates can be FREE, CHEAP, or EXPENSIVE

Standard Template Library

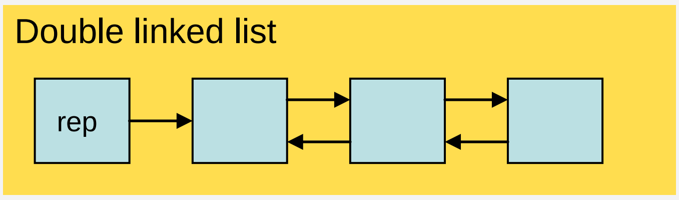
A part of C++ is the Standard Template Library (STL). As its name suggests, this library provides a resource of pre-composed templates that can be used to easily implement data structures, such as stacks, lists and queues and the algorithms operating on them.

Many of the templates use the concept of containers, which is illustrated below:

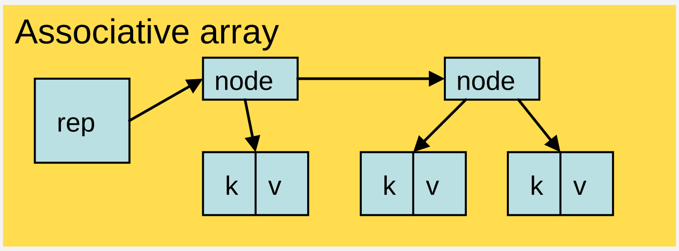
```
// Vector of ints
vector<int> vec;
```



```
// List of chars
list<char> li;
```



```
// Map with char keys
// and float values
map<string, int> phonedir;
```



Example 11: Containers; Some examples of how C++ uses the concept of containers to implement data structure

Containers use their own version of pointers to refer to a specific element. The iterator is a smart pointer, and has different features for different containers. The iterator to a vector, for example, can be randomly moved within the vector, while that for a list can only move one step forward or backwards.

```
vector<int> v;

v[0] = 21;
v[8] = 1;

vector<int>::iterator b = v.begin();
vector<int>::iterator e = v.end();
vector<int>::iterator i = b + 3;

*i = 7;

// sort(i, e);
```

Example 12 (a): STL vector. This will cost 1000 bytes without sort(), which will add a further 2300 bytes of code.

```
map<string,int> m;
int x;

m["monday"] = 1;
m["tuesday"] = 2;
m["wednesday"] = 3;
m["thursday"] = 4;
m["friday"] = 5;
m["saturday"] = 6;
m["sunday"] = 7;

x = m["friday"];
```

Example 12 (b): STL map. This will cost 7000 bytes. Replacing string with char will reduce the price to 5500 bytes. However, additional use of map is significantly cheaper, as the code is reused. Another map of the same type adds only 100 bytes, while using a different type of map adds 2000 bytes.

In STL there are a great number of algorithms for operating on the iterators. There are three significant groups that do not modify the contents of the containers, such as `for_each`, `find`, and `count`, those that do modify the contents, such as `transform`, `copy`, `replace`, `fill`, `generate` and `remove` and those that sort the contents, such as `sort`, `lower_bound`, `binary_search` and `merge`.

Using STL reduces implementation time, compared with hand crafting, and the result is likely to work as intended but at a considerable code and data cost. Using the STL library is EXPENSIVE.

RTTI

RTTI, RunTime Type Information, allows a running application to find out the identity of derived classes, either by asking for the name using `typeid`, or by checking if the class is of the expected type using `dynamic_cast`. It requires the literal names of all classes to be part of the application binary, and also adds extra code, it is EXPENSIVE.

Exceptions

The C++ exception-handling mechanisms are provided to report and handle errors and exceptional events. A function that finds itself in a situation that can not be handled by a standard return, can `throw` an exception. A function higher up in the call chain can register to `catch` that exception. A lot of extra code is needed to implement the exception mechanism, it is EXPENSIVE.

SUMMARY

The most expensive elements of C++, when compared with C code, is STL and exceptions. Templates themselves are very implementation dependent and they can range from expensive down to free. Classes, namespaces, inlining, operator overloading, constructors/destructors and references are all effectively free when compared with C for the same result and virtual functions incur a small expense but are still relatively cheap.

The cost in terms of code size overhead compared to C for the C++ constructs we have discussed are

- encapsulation/classes - FREE
- namespaces - FREE
- inlining - FREE
- operator overloading - FREE
- constructors/destructors - FREE
- references - FREE
- virtual functions - CHEAP
- templates - EXPENSIVE
- STL (Standard Template Library) – EXPENSIVE
- RTTI – expensive
- exceptions - expensive

But the real cost, or lack of it, for C++, can often come from the code writer. Perhaps even more than some other languages, the power and options available within C++ can lead to one implementer producing tight, “cheap” code, while leading another astray to produce a much more expensive result.