Inefficiencies of C++: Fact or Fiction?

Shawn A. Prestridge, Senior Field Applications Engineer

Agendum

- Why even consider C++ in an embedded system?
- The C++ pricing structure
- Encapsulation: Classes and Namespaces
- Implicit inlining
- Operator overloading
- Constructors/destructors
- References
- Inheritance/virtual functions
- Templates
- RTTI and Exceptions
- Summary

Why use C++ in an embedded system?

What's the deal?

- According to Stroustrup, "C++ is a general-purpose programming language designed to make programming more enjoyable for the serious programmer."

- What you don't use, you don't pay for

The C++ Pricing Structure

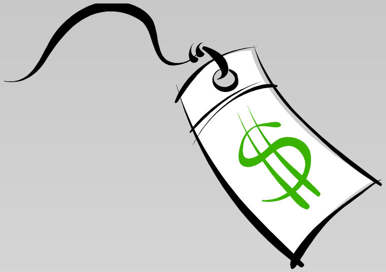
How much does it cost?

The price tags:

- Free – no overhead compared to coding in C

- Cheap – Small overhead compared to coding in C

- Expensive – Large overhead compared to coding in C



* Note that the pricing structure on constructions is based on the Embedded Workbench, so pricing may vary on other tools

Encapsulation

Encapsulation of data and code

- Allows you to do implementation/information hiding
- There are two basic constructions in C++ that allow us to do this:
    - Classes (main technique for encapsulation)
    - Namespaces

Example of a class

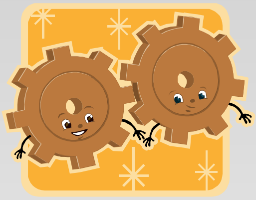
```
class CircularBuffer
{
private: // implicit private
  unsigned char mBuffer[256];
  unsigned char mFirst;
  unsigned char mLast;

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();
};
```

What’s under the hood of a class?

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

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

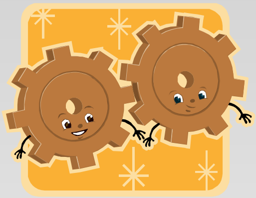


Using a class

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

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

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



Price of a class

- The function call is made in the same way as C; name mangling is used to make the function name unique to the linker
- The cost of setting up the pointer to the object and passing it to the member function is likely to be done in similar ways as in a C program
- Cost: FREE

Namespaces

- Namespace is a mechanism to group all visible names together within that namespace
- Code outside the namespace must qualify with the namespace name to refer to data within the namespace, for example Decoders::bitrate
- Cost: FREE


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



if (Decoders::bitrate == 192)
```

Implicit Inlining

Inlining your function call

- Sounds expensive
- Small inlined member functions are cheaper compared to no inline; no function call overhead
- Cost: FREE


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

Operator overloading

Operator overloading

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

CircularBuffer buf, buf_a, buf_b;

buf = buf_a + buf_b;
```

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

- Convenient way of defining meaning of the standard operators +, -, |, ... for your class
- Cost: FREE

Constructors/destructors

Constructors/destructors

- Constructors/destructors are called implicitly when an object is created
- No extra code is generated other than as seen in the constructor/destructor code.
- Cost: FREE


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

References

References

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

```
MOV     R1,#+5
STR     R1,[R0, #+0]
```

- Mostly used for parameters
- Same cost as passing a pointer

- Cost: FREE

Inheritance/virtual functions

Inheritance and virtual functions

- Track specifies the interface between a Track and the rest of the world
- Other classes will inherit from Track
- = 0 means that the function must be defined in the derived class
- The class is abstract and no objects of type Track can be created

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

Inheritance and virtual functions

```
class Mp3Track : public Track // derived class
{
public:
  // Extract Artist and Title info from ID tag
  virtual string const & Artist();
  virtual string const & Title();

  // Play the audio data
  virtual void Play();
};

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

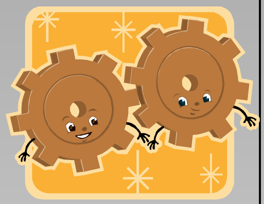
- Mp3Track must implement all functions in Track with "= 0"
- WmaTrack will look similar but decodes WMA instead of MP3, ditto for OggTrack
- Separating interface and implementation means that we don't need to care what type of track we are dealing with in DoMusic
- p->Play() will call Mp3Track::Play() or WmaTrack::Play() depending on which Track implementation p points to

Inheritance and virtual functions

```
void Mp3Track_Play(struct Mp3Track
  *this);

typedef void (*Fptr)(struct
  Mp3Track *);
typedef Fptr (VTable)[3];

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



```
struct Track
{
  // This is invisible
  VTable const *vptr;
};

struct Mp3Track
{
  struct Track mBase;
};

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

```
LDR     R1,[R0,#+0]
LDR     R1,[R1,#+8]
BLX     R1A
```

Inheritance and virtual functions

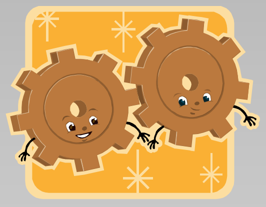
- Comparing ARM architectures
- ARM, Thumb, and Thumb2 modes

ARM7 (v4) arm

```
LDR     R1,[R0,#+0]
LDR     R1,[R1,#+8]
MOV     LR,PC
BX      R1
```

ARM9E (v5)

```
LDR     R1,[R0,#+0]
LDR     R1,[R1,#+8]
BLX     R1
```



ARM7 (v4) thumb

```
LDR     R1,[R0,#+0]
LDR     R1,[R1,#+8]
BL      _bx_r1


_bx_r1:
BX      R1
```

Cortex-M (7m)

```
LDR     R1,[R0,#+0]
LDR     R1,[R1,#+8]
BLX     R1
```

Inheritance and virtual functions

The same thing can be done in C in many different ways:

- `switch` statement on TrackType
- Table lookup and function call through a function pointer
  - This is needed for all functions, including Artist, Title, and Play
  - Instead of one vtable per class there will be a table of function pointers per function

Drawbacks with doing it in C:

- The track details tend to be spread all over the code
- Makes it non-trivial to add support for a new track type, e.g. AAC audio format

Inheritance and virtual functions

The cost:

- One vptr per object
- One vtable per class with virtual functions
- Calls to virtual functions must follow vptr and lookup the function address in vtable (in C, the function address can be looked up in a table without following a pointer)
- 4 bytes extra per created object compared to a table in C
- Constructor code to setup vptr and vtable data

Cost: CHEAP (almost free)

Templates

Templates - functions

```
/* 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.
}
```

- Function templates are in some ways a substitute for macros. The advantage with function templates is that they are more secure syntactically and semantically.
- As with macros, each invocation potentially generates some code.

Templates - class

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

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

- Creates a class based on int with 6 as constructor parameter
- Class templates can have function templates
- With templates, you can build rather complex structures.
- With complexity comes the potential for hidden code size cost.

Templates - example

```
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;
```

- Clever use of templates can compute values and catch errors at translation time rather than run-time
- No code is generated, only the constant 24

Templates - cost

- Template complexity can vary greatly
  - Simple macro expansions
  - Complex expansions producing lots of code
- Cost: It depends, FREE - EXPENSIVE

Avoid "overly clever" template meta-programming

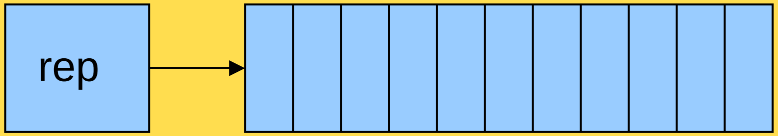
STL – Standard Template Library

- Library of containers.
- Implements commonly used data structures, and algorithms that operate on them.
- Built on the C++ template mechanism.

STL - containers

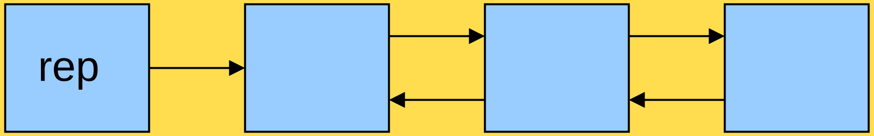
```
// Vector of ints
vector<int> vec;
```

Dynamic C array



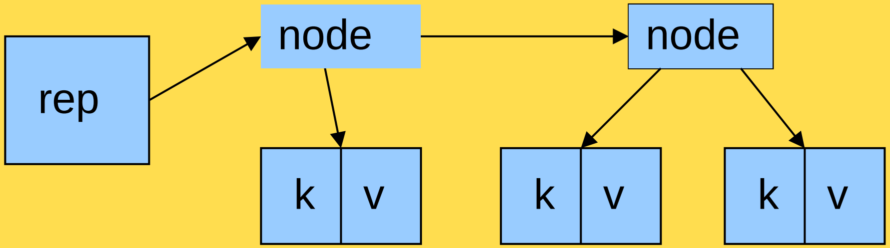

```
// List of chars
list<char> li;
```

Double linked list




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

Associative array



STL - iterators

- When referring to a specific element in a container, an iterator is used; essentially, that is a pointer to the specific element, albeit a very smart pointer.

- Each container has its own kind of iterator with different kinds of features:
  - An iterator into a vector can be randomly moved inside the vector,
  - An iterator into a list only can move one step forward or backward, etc.

STL - vectors

```
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; // i points to v[3]
 // sort(i, e);
```

- Cost: 1000 bytes
- The `sort()` method adds another 2300 bytes

STL - maps

```
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"];  //x is assigned the value 5
```

- Cost: 7000 bytes (5500 if using char* instead of string)
- Additional use of map is cheaper, code is reused:
  - Additional map of the same type adds just 100 bytes
  - Additional map of a different type adds another 2000 bytes

STL - algorithms

There are quite a few algorithms defined by the STL that operate on iterators

Non-modifying:

- `for_each`
- `find`
- `count`

Modifying:

- `transform`
- `copy`
- `replace`
- `fill`
- `generate`
- `remove`

Sorting:

- `sort`
- `lower_bound`
- `binary_search`
- `Merge`

STL - cost

- STL uses the heap (`new`/`delete` – `malloc`/`free`)
- The benefit of using the STL containers instead of your own handcrafted container is that you can be pretty sure that the STL versions work as intended and you gain the implementation time.
- Cost: EXPENSIVE

RTTI and Exceptions

RTTI and Exceptions

```
void Status(Track* p)
{
  type_info &info = typeid(p); //class name in info.name()

  // non-NULL if p is of type Mp3Track
  Mp3Track* mp3ptr = dynamic_cast<Mp3Track*>(p);
}
```

- RTTI 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.
- Cost: EXPENSIVE

RTTI and Exceptions

```
void FuncA(void)
{
  try
  { FuncB(); }
  catch (int e)
  {
    …
  }
}
```

```
void FuncB()
{
  FuncC();
}
```

```
void FuncC()
{
  if(error)
    { throw 23; }
}
```

- 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.
- Cost: EXPENSIVE

Summary

SUMMARY

Free:

- Classes
- Namespaces
- Inlining
- Operator overloading
- Constructors/desctructors
- References

Expensive:

- STL
- RTTI
- Exceptions

Cheap:

- Virtual functions

Free-Expensive (it depends):

- Templates

Caveats – differences in C and C++

- `static int i = f(6);`
- `const int i = 6;`
- `volatile int j;`

```
void f()
{ j; }
```

- Complex initializers for static variables done at runtime
- Constants are global in C but static in C++ (need keyword extern for global)
- Volatile variables are rvalue in C, lvalue in C++ which means no access in C++

THANK YOU FOR YOUR ATTENTION!