C++FA 3.1 OPTIMIZING C++

Ben Van Vliet

Measuring Performance

Performance can be measured and judged in different ways—execution time, memory usage, error count, ease of use—and trade offs usually have to be made. In quantitative finance and automated trading, execution speed is usually the most highly sought-after quality. Thus, being able to measure execution time is important. We express execution time as a function of the number of data items (N) that a routine manipulates over its run. We use the results as estimates in O notation, as in O(N), O(log N), etc. These estimates enable us to project the distribution of performance under different conditions.

To actually measure performance in given conditions, we need to measure time. This is logically done:

- Get the system time as the starting time.
- Execute some C++ routine.
- Get the system time as the ending time.
- Execution time is the ending time minus the starting time.

We may need to execute a routine many, many times in order to get a resolution long enough to measure. The following is a Timer class that will help in measuring elapsed time.

```
#include <ctime>

class Timer
{
private:
	clock_t startTime;
	clock_t stopTime;

public:
	void start()
	{
		startTime = clock();
	}
	void stop()
	{
		stopTime = clock();
	}
	double elapsed()
	{
		return ( stopTime - startTime ) / 1000.0f;
	}
};
```

```
int main()
{
	Timer t;
	t.start();

	// Execute some routine here.

	t.stop();
	cout << t.elapsed() << endl;
	return 0;
}
```

The resolution of clock() in C++ is not very good. To build a timer class with greater granularity, say down to the microsecond, you will need library classes particular to your operating system (e.g. Windows API).

Improving Performance

In this section, I summarize 20 well-known categories of techniques for optimizing C/C++ code. These are especially useful when developing real-time systems, where execution speed is imperative. While they are fairly straightforward, these techniques can nevertheless sometimes result in big performance improvements. (You should be able to design test programs and measure performance improvements using the Timer class.) Before I get started, though, let's review a few important concepts:

- The fastest code is code that never runs. A complement to this rule is, the only code that is error free, is code that is never tested. Worry about the design and the correctness of the code first, the performance of the code last. Fast code is no good if it produces an incorrect result.
- Notwithstanding the previous point, you can apply many of the optimization techniques in this paper as you code.
- Reducing the number of lines of code will not necessarily improve performance. Sometimes, adding code improves performance.

Also, remember that good software design and project management practices will not only speed development, but also produce better code. As with all software development processes, I recommend using an iterative approach to planning, designing, developing and optimizing code. Specifically, development teams should define and document up-front:

- Software attribute requirements for performance. For real-time systems, this may include specification of how responsive a system should be and measurement of how responsive it is. This can also apply to target frame rates, latency and bandwidth.
- System architecture. For real-time systems, hardware considerations will impact coding decisions, including the operating system, the number of processors, the number of cores and number of threads being used. Further, the object oriented design, reusability issues, and the use of external components, such as databases or third party software, will impact coding decisions.

When optimizing code, benchmark everything using a profiler, including alternative algorithms (e.g. <stdio> functions are faster than <iostream> ones, i.e. printf is faster than cout) and external, COTS components. STL containers are great, but better performance may be had by writing your own—a doubly-linked list is overkill and will perform slowly relative to a singly-linked list, if insert-at-front is all you need. You may also consider using one of the many other containers out there, such as CRC32, MD5, HAVAL 128, Tiger, ADLER32, PERFECT Hash, Sparse, Dense, et al.

So, here are the 20, in no particular order.

1. Optimize declarations.

The lower the number of local variables in a function, the better the compiler will be able to fit them into registers. Thus, the compiler will avoid performing pointer frame operations on local variables on the stack. If all local variables are in registers, performance will be better than accessing them from memory. If no local variables are present, the compiler will avoid the overhead of frame pointer set up and restoration.

Declare variables with the inner-most scope possible. You will get better performance if local variables are declared only in cases where needed, rather than in every case. Put declarations inside if statements if possible.

```
// This…
int foo( int &x )
{
    if ( x > 0 )
    {
       int y = 7;
       return x + y;
    }
    return x;
}

// …is faster than this
int foo( int &x )
{
    int y = 7;
    if ( x > 0 )
    {
        return x + y;
    }
    return x;
}
```

The opposite goes for loops. A declaration before a loop happens only once. A declaration inside a loop causes a new construction each iteration. If possible, put declarations outside the loop.

The processor keeps referenced data and function code in a cache so the next it is referenced, it gets it from the cache. Such cache references are faster. Try to place function code and data that are used together, together physically. In C++ objects enforce this. In C, arrange the declaration order of related, or closely coupled, data and functions so as to be close together.

In the ideal case, postpone declarations until you can initialize the value.

```
// This…
int x = 5;

// …is faster than this
int x;
x = 5;
```

Initializing can be up to four times faster than declaring and then assigning a value. Even for strings, the performance gain can be significant.

2. Optimize function parameters.

Calls to functions with several parameters can be expensive, especially when using pass by value since each parameter must be pushed on the stack. For similar reasons, pass reference whenever possible, especially for types bigger than 4 bytes. For example,

```
// This…
template < class T >
void foo( T &x )
{}

// …is faster than this
template < class T >
void foo( T x )
{}
```

Passing parameters by value is fine for regular types like integer, pointer, etc. When passing bigger types, the cost of copying the object on the stack can be prohibitive. In case of passing objects there is the additional overhead of invoking the copy constructor for the temporary copy that is created on the stack. Always pass objects by reference. Existing pass-by-value methods can be easily safely modified to pass-by-reference by simply using const references. Const will prevent any changes to the object over the course of the method definition.

With respect to pass-by-pointer versus pass-by-reference, consider using pass-by-reference. For example,

```
int PassByPointer( int *x ) { return *x + 5; }
int PassByReference( int &x ) { return x + 5; }
```

When compiled, these two functions generate exactly the same machine language code. The advantage of the PassByReference version is that there's no need to check that the reference is not NULL (since references can never be NULL) as there is with the pointer.

Because a compiler cannot know when different pointers refer to the same location, it cannot produce the fastest possible code. Since the value of a reference is static, the compiler can better-optimize the machine code. In almost all cases, pass-by-reference performs significantly faster.

3. Optimize return values.

A function does not know (and does not care for that matter) if a return value is used. It will always pass the return value if one is defined. It’s obvious, but if it’s not going to be used, do not define a return value.

Passing bigger types as return values has the same performance issues as bigger parameter types. Returning an object by value requires that the entire object be copied, whereas returning a reference will not force a call to the copy constructor.

```
// This is slow.
template< class T >
T ReturnByValue()
{
    T m_Obj;
    return m_Obj;
}
```

While returning an object in this method is safe, its slow.

```
// This is better.
template < class T >
T &ReturnByReference()
{
    T m_Obj;
    return m_Obj;
}
```

It’s best return an object value via a non-const reference parameter.

```
// This is better still.
template< class T >
void PassReturnTypeAsParameter( T &m_Obj )
{
    m_Obj.Foo();
}
```

4. Create structures with sizes that are powers of 2.

When arrays of structures are involved, the compiler multiplies by the structure size to perform the array indexing. If the total structure size is a power of 2, an expensive multiply operation will be replaced by an inexpensive shift operation, which will improve performance in array indexing.

5. Optimize switch…case statements.

When it comes to switch…case statements, put the most common cases first. For big switch…case statements, relegate unlikely cases to a smaller nested switch…case, which is the default leg of the larger switch…case. Placing the most frequently hit labels first reduces the number of comparisons that will be performed on most occasions.

```
switch ( hair_color )
{
     case BROWN:
          break;
     case BLONDE:
          break;
     case RED:
          break;
     default:
          switch ( hair_color )
          {
               case PINK:
               case PURPLE:
               case BLUE:
          }
}
```

6. Optimize constructors.

As much as possible, keep constructors simple. The constructor is invoked every time an instance of the class is created, and the compiler might be creating temporary object over and above your explicit instantiations. Optimizing the constructor will give you a performance boost.

In the constructor, use initialization lists to set the class member variables. Assignments within the constructor body gives lower performance.

```
// This…
template < class T >
MyClass::MyClass( const T &v ) : value( v )
{
}

// …is faster than this.
template < class T >
MyClass::MyClass( const T &v )
{
         value = v;
}
```

A member variable's default constructor is called anyway before the class' constructor, so assign the value then. A drawback to initialization lists is that no error checking is possible.

An object with one-stage construction is one where the fully instance is fully built by the constructor. An object with two-stage construction is built only minimally in the constructor, and fully built later on using class methods. If your going to copy an object frequently (and you shouldn't), an expensive constructor can impact performance. Consider two-stage object construction instead.

Lastly, use explicit constructors. The explicit keyword avoids the problem where a single argument constructor causes the compiler to perform an automatic type conversion. Declare all single-argument constructors as explicit.

7. Minimize the use of virtual methods.

Virtual function calls are more expensive than regular function calls, so avoid declaring methods as virtual unless you know they will be overridden. It's easy to add the virtual specifier later on if need be.

The presence of even a single virtual method means that every instance of the class and every instance of a derived class will contain an extra pointer, and this additional level of indirection means that virtual method calls take longer. The following example shows that a small object can double in size due only to virtual methods.

```
// sizeof MyClass is 8 bytes
class MyClass
{
private:
   int value;
public:
   MyClass() { value = 0; }
   virtual ~MyClass() {}
   virtual int get_value() { return value; }
};

// sizeof MyClass is 4 bytes
class MyClass
{
private:
   int value;
public:
   MyClass() { value = 0; }
   ~MyClass() {}
   int get_value() { return value; }
};
```

Creating instances of virtual classes is also more expensive because the compiler has to initialize a table of virtual methods.

8. For small functions, declare them as inline.

Declaring functions as in line, best for small functions of 1-3 lines of code, removes the overhead of a function call, passing of parameters, stack frame manipulation and returning values, especially big values. For bigger functions, in line can actually have a negative impact on performance. Duplicate function definitions for large functions embedded in line will bloat your code, and may cause cache misses which will result in slower execution. Of course, in line only asks the compiler to consider a function for in line expansion. So, there's no guarantee that it will actually happen.

A good profiler should be able to tell you which functions would benefit from in line.

9. In for loops, count down rather than up.

When it comes to for loops, count down to zero rather than up if possible. The test against zero is done every iteration and it's faster than any other test.

```
// This…
for( int x = 99; x > 0; --x )

// …is faster than this
for( int x = 1; x < 100; ++x )
```

10. Where possible, use prefix operators rather than postfix.

In the previous example, the prefix operation ++x is faster than would be the postfix x++. With prefix, the value is increased and the new value returned; with postfix, the value is increased, but the old value returned, so postfixing requires that the original value be saved in a temporary object. Avoid postfixing.

11. Use operator=, rather than just the operator.

```
//This…
x += 3;

// …is faster than this
x = x + 3;
```

The first version is better than the second because, again, it avoids creating a temporary object.

For overloaded operators, it's generally more efficient to use the += instead of + , because again += won't cause generation of a temporary object.

12. Use nameless objects.

```
// This…
m_List.push_front( MyClass( 3 ) );

// …is faster than this
MyClass m_Obj( 3 );
m_List.push_front( m_Obj );
```

In the first case, the parameter and the object share memory. This holds true for return values as well.

13. Minimize the use of exception handling.

Exception handlers are great, but they're expensive. Programs using exception handling can be larger and slower. Avoiding using try…catch blocks except where necessary. I believe the judicious use of exceptions is the best solution. Also, use the throw() function exception specification to tell the compiler about functions that don't throw exceptions.

14. Avoid type checks during runtime.

Using runtime type identification you can get information about objects while the program is executing. The dynamic_cast operator relies on runtime time identification to perform the proper casting. For this to work, a program has to save information about every class that contains virtual functions. This makes your program larger. Avoid the dynamic_cast and typeid operators.

15. Use the integer data type if at all possible.

It should go without saying that care should be taken to select the proper data type—integer or float, list or vector, hash_map or tree. Using the right type can make a difference. That having been said, try to use the `integer` data type wherever possible. Integer is always the native type for any machine and C++ performs arithmetic operations and parameter passing at integer level. If, for example, you use a char to hold an integer value, the compiler will first convert the values into integers, perform the operations and then convert back the result to char. Avoiding conversions will save time.

16. Make local functions static.

Always declare local functions as static. This means they will not be visible to functions outside the .cpp file, and some C++ compilers can take advantage of this in their optimizations.

17. Optimize if statements.

```
// This…
bar();
if ( y > 0 )
{
    undo_bar();
    foo();
}

// …is faster than this
if ( y > 0 )
{
    foo();
}
else()
{
    bar();
}
```

Use a profiler and good judgment to decide if undoing the bar() operation is faster than jumping.

18. Avoid the use of expensive operations.

Addition is cheaper than multiplication and multiplication is cheaper than division. Factor out expensive operations wherever possible. It goes without saying that expensive operations should be avoided.

19. Advanced Methods

Consider per-class allocation. C++ allows us to overload new and delete at the class level. For some objects, this can generate speed improvements. As an example, the memory pool method, which uses pools for memory allocation, improves locality relative to new. If new or delete are causing bottlenecks, consider overloading them.

Also try something called template metaprogramming, a technique that uses templates as pre-compilers. By embedding a call to a template within itself, you can force the compiler to recursively generate code. A good compiler will optimize the recursions away since all template parameters must be known at compile time. This can result in performance improvements. Template metaprogramming is effective especially for quant libraries though results will vary by compiler. Here is a recursive factorial function done the normal way.

```
int factorial( int x )
{
     if ( x == 0 )
          return 1;
     return x * factorial( x – 1 );
}
```

Now, lets take a look at a template-based version that calculates the factorial, where x is the template parameter, not the function parameter.

```
template< int x >
struct Factorial
{
     enum
     {
          value = x * Factorial< x - 1 >::value
     };
};

template <>
struct Factorial< 0 >
{
     enum { value = 1 };
};

int main()
{
      int a = Factorial< 5 >::value;
      cout << a << endl;

      return 0;
}
```

The template calculates the factorial at compile time; the result is a precalculated constant. To use templates this way, the compiler must know the parameter values at compile time. Which is to say that the parameter x must be a constant literal or constant expression.

Finally, try copy-on-write where two or more objects share the same data, at least until one of the objects needs to be changed. Then, the data can be copied and changed. Copy-on-write requires reference counting of the number of objects referring to the same data and smart pointers, which point to objects with a reference count. When the reference count is zero, a smart pointer automatically deletes the object. When a smart pointer is copied, the reference count is incremented; when it's destroyed the reference count is decremented. Performance improvements using copy-on-write will depend on how often objects are copied (which you should avoid in the first place) and how often those objects change.

20. Compiler optimizations are also key.

A good compiler will generate faster, better optimized code though optimization may not necessarily happen on its own. Compilers usually produce consistent and correct code rather than performance-optimized code. Most compilers have settings, or flags, which allow you to direct it toward performance optimization and you should investigate these settings for your compiler.

Conclusion

So, we have looked at 20 classifications of common techniques for optimizing C++ code. But, we are not done! This is only the beginning. A good programmer's mantra is: refactor, refactor, refactor. Your firm's internal improvement processes should require a continuous search for new optimizations, especially when new technologies become available—new hardware, new compilers, etc. Of course, refactoring has tradeoffs and where speed of execution is mission critical, refactoring for performance optimization can sometimes make code less modular, and therefore more difficult to debug and maintain. As always, you should use a good profiler to guide you.