7COM1025
Programming for Software Engineers
Lecture 19

Dr. Renato C. de Amorim

REQUIRED READING

This class relates to chapter 7 of the book:

Reddy, M. (2011). API Design for C++. Elsevier.

Pages: 209 - 240

PERFORMANCE

Your API should be as optimal as possible so that it does not actively undermines performance.

Good API designs normally correspond with good performance.

Components of API performance:

Compile-time speed: The impact your API has on the time it takes to compile a client program.

Run-time speed: The time overhead for calling your API methods.

Run-time memory overhead: The memory overhead for calling your API.

Library size: The size of the object code for your implementation. This affects the total disk and memory footprint of your client's applications.

Startup time: The time it takes to load and initialise your API.

PERFORMANCE

Remember:
The overall performance improvement gained by optimising a single part of your system is limited by the fraction of time that the improved time is actually used.

You may improve a particular function speed by a factor of 10, but if this function is used 1% of the time then the overall improvement is of 0.1 (10 * 0.01).

RECOMMENDATIONS (CONST REF)

Pass input arguments by const reference.
It passes a reference to the object (no call to the copy constructor) and doesn't allow the object to be modified

This recommendation applies to objects, and not really to built-in types (int, bool, float, etc). These are small enough to fit in the CPU register. STL iterators are designed to be passed by value.

Note: If a method accepts a base class argument by value and you pass an object of a derived class, then the extra fields of the derived class will be "sliced off"

RECOMMENDATIONS (#INCLUDE)

Minimise #include dependencies.
The time it takes to compile a large project can depend greatly on the number and depth of #include files.
Note this relates to the size of these files.

It is better to #include 2 small files than to #include one large file that has everything you need – and more.

Be careful with redundant #include. You can deal with this using the preprocessor:

```
#ifndef HEADERFILE_H
#define HEADER_FILE_H
//Code
#endif
```

And include this in the .cpp as

```
#ifndef HEADERFILE_H
#include <headerfile>
#endif
```

RECOMMENDATIONS (INITIALIZATION LISTS)

Member variables are constructed before the body of the constructor is called.

```
Class myclass{
private:
      string FirstName;
      string LastName;
public:
      myclass(const string &first, const
string &last)
{
      FirstName = first;
      LastName = last;
}
};
```

```
class myclass{
private:
      string FirstName;
      string LastName;
public:
      myclass(const string &first, const
string &last) : FirstName(first),
LastName(last){}

};
```

Left: the constructor from string will be called, and then inside the constructor the assignment operator will be called.
Right: Only the copy constructor is called.
Note: You can declare the constructor as myclass(const string &first, const string &last);
And define it (in the .cpp) as   myclass(const string &first, const string &last) : FirstName(first), LastName(last){}

RECOMMENDATIONS (CONSTANTS)

Use constants instead of hardcoded values.

Issue: Any variable that you define in this way will cause your compiler to store space for the variable in every module that includes your header.

Solution: declare them using extern
extern const in var;

RECOMMENDATIONS (MEMORY)

On modern CPUs memory latency can be one of the largest performance concerns.

Speed of processors increases why faster than the speed of DRAM

Remember:
- The smaller your objects are, more can be put in the cache.
- cluster variables by their type. Computers use a “word” at a time, this helps to avoid wasting some of these (check section 7.5 of the book).
- You can use unions for variables to share the same area of memory:
union { float floatvalue; int intvalue;} FloatOrIntValue;
- Don't add virtual methods until you need them.
- Don't allocate memory until you need to.
- Deallocate objects you won't need anymore.

RECOMMENDATIONS (ITERATORS)

Iterators can transverse over some or all elements in a container class. It's a well-known pattern.

They can be used to transverse massive data sets that may not even fit entirely into the memory.

Clients may create multiple iterators to transverse the same data simultaneously.

Random access

For random access, overload []. This is normally done without bounds checking so that it can be very efficient.

You can create a .at(). This method should throw an exception if the index is out of range.