Lecture 11a
Efficiency

April 4th, 2016

Announcements

- No reading due this or next week
- Recitations: doxygen
- PA7 is due Wednesday
- Midterm 1 week from Wednesday
- Exam week
  - Monday: review + hand out code
  - Wednesday: midterm #2
  - Friday: midterm post-mortem

Efficiency

- Efficiency is a topic skipped by your book
  - Odd, since it is a major reason for using C++ 😊
- We have alluded to it
  - But not focused on it
  - Have not graded on it
- Instead we have focused on quality
  - How to write clean, well-designed code
  - How to test code for correctness
- There is a reason for this...

“Premature optimization is the source of all evil”
--Donald Knuth

- Focus on quality first
  - Write clean, safe code
    - E.g. use foo.at(i), not foo[i]
  - Test your code
    - Will serve as reference implementation
  - Efficiency comes last
    - Ask yourself: how efficient does it have to be?

Efficiency Process

1. O() analysis
   - Source of largest improvements
   - Consider how/whether your data will grow
     - For example, in our task
       - The number of points per pose is fixed
       - The number of poses per file is open ended

2. Profile
   - Use typical inputs, if possible
   - Find out where your time is going

3. Attack the hot spots
   - 10% of code typically accounts for 90% of runtime
   - Don’t optimize most of our code
   - Just look at the routines that are consuming time
   - Yet another reason for small methods/functions

From last Friday:
Informal Assignment for Monday

- Run the valgrind profiler on your PA6
- Note where the time goes
- Be prepared to tell me (and the class) on Monday

So where did your time go?
What are your bottlenecks?
List of Bottlenecks:

Previous Bottlenecks

• Copying on function calls
  – Use const pass by reference!
• Copying on update
  – For example, when normalizing poses
  – Update in-place
• Unnecessary loops
  – Loop over data more often than needed
• I/O
  – read entire file into char array
  – Use istringstream to parse it
  – Avoids OS calls
  – fseek / ftell

More Previous Bottlenecks

• Array/vector accesses
  – Reserve room for vectors in advance
  – [] more efficient than at()
  – Iterators and/or pointer hopping
• Integers are faster than floating point
• Inline your methods if they are short enough
  – But only very short methods that are called often!
• Double initialization
  – Use initializer lists
• Failure to release memory
  – Don’t store more data in main than necessary
  – Don’t hold onto data longer than necessary
  – Avoid global variables

Pointer Hopping

• Less efficient:
  // arr is a (simple) array
  for(int i=0; i < size; i++) {
    sum += arr[i];
  }
• More efficient:
  int* end = arr + size;
  for(int* ptr = arr; ptr < end; ptr++) {
    sum += *ptr;
  }

Iterators

• Less efficient:
  // vec is a vector<int>
  for(int i=0; i < size; i++) {
    sum += vec[i];
  }
• More efficient:
  vector<int>::iterator iter = vec.begin();
  vector<int>::iterator end = vec.end();
  for(; iter < end; iter++) {
    sum += *iter;
  }

Return Value Optimization

• Least Efficient
  Quagga q;
  q.name = "fred";
  return q;
• More efficient
  Quagga q("Fred");
  return q;
• More efficient still
  return(Quagga q("Fred");)
• This allows the compiler to make q in the memory of the receiving function...
Other Hints

• Compiler optimization flags
  – Let the compiler help you!
  – -O2 is fast, safe.
  – -O3 is potentially risky
  – Works better, the more you use const

• Minimize memory consumption

• Keep your reference implementation
  – Compare results
  – It’s easy to insert bugs while optimizing!