Biostatistics 615/815 Lecture 4: Classes and Libraries, and Divide and Conquer Algorithms

Hyun Min Kang

September 13th, 2012

Recap - Call by value vs. Call by reference

callByValRef.cpp

```
#include <iostream>
int foo(int a) {   // a is an independent copy of x when foo(x) is called
  a = a + 1;
  return a;
}
int bar(int& a) {  // a is an alias of y when bar(y) is called
  a = a + 1;
  return a;
}
int main(int argc, char** argv) {
  int x = 1, y = 1;
  std::cout << foo(x) << std::endl; // prints ??
  std::cout << x << std::endl;      // prints ??
  std::cout << bar(y) << std::endl; // prints ??
  std::cout << y << std::endl;      // prints ??
  return 0;
}
```


Recap - Precision for very large values

intFac() - only calculates up to 12!

```
int intFac(int n) {  // calculates factorial
  int ret;
  for(ret=1; n > 0; --n) { ret *= n; }
  return ret;
}
```

dblFac() - calculates up to 170!

```
double dblFac(int n) { // main() function remains the same
  double ret; // use double instead of int
  for(ret=1.; n > 0; --n) { ret *= n; }
  return ret;
}
```


Recap - Precision for very large values

logFac() - Allows much larger range

```
double logFac(int n) {
  double ret;
  for(ret=0.; n > 0; --n) { ret += log((double)n); }
  return ret;
}
```

Improving Time Complexity

logFac() : $\Theta(n)$ of log() calls

```
double logFac(int n) {
  double ret;
  for(ret=0.; n > 0; --n) { ret += log((double)n); }
  return ret;
}
```

Fisher's Exact Test requires : $\Theta(n^2)$ of log() calls

```
  for(int x=0; x <= n; ++x) { // among all possible x
    if ( a+b-x >= 0 && a+c-x >= 0 && d-a+x >=0 ) { // consider valid x
      double l = logHypergeometricProb(x,a+b-x,a+c-x,d-a+x);
      if ( l <= logpCutoff ) pFraction += exp(l - logpCutoff);
    }
  }
```


Precomputing factorials reduces log() calls to $\Theta(n)$

function initLogFacs()

```
void initLogFacs(double* logFacs, int n) {
  logFacs[0] = 0;
  for(int i=1; i < n+1; ++i) {
    logFacs[i] = logFacs[i-1] + log((double)i); // only n times of log() calls
  }
}
```

function logHyperGeometricProb()

```
double logHypergeometricProb(double* logFacs, int a, int b, int c, int d) {
  return logFacs[a+b] + logFacs[c+d] + logFacs[a+c] + logFacs[b+d]
    - logFacs[a] - logFacs[b] - logFacs[c] - logFacs[d] - logFacs[a+b+c+d];
}
```


C++ class example : Point

```
#include <iostream>
#include <cmath>
class Point {
public:
   double x, y;  // member variables
   Point(double px, double py) { x = px; y = py; } // constructor
   double distanceFromOrigin() { return sqrt( x*x + y*y ); }
   double distance(Point& p) { // distance to another point
     return sqrt( (x-p.x)*(x-p.x) + (y-p.y)*(y-p.y) );
   }
   void print() { // print the content of the point
     std::cout << "(" << x << "," << y << ")" << std::endl;
   }
};
int main(int argc, char** argv) {
  Point p1(3,4), p2(15,9);   // constructor is called
  p1.print();                                  // prints (3,4)
  std::cout << p1.distance(p2) << std::endl; // prints 13
  return 0;
}
```


C++ class example - Rectangle

```
class Rectangle { // Rectangle
public:
  Point p1, p2; // rectangle defined by two points
  // Constructor 1 : initialize by calling constructors of member variables
  Rectangle(double x1, double y1, double x2, double y2) : p1(x1,y1), p2(x2,y2) {}
  // Constructor 2 : from two existing points
  Rectangle(Point& a, Point& b) : p1(a), p2(b) {}
  double area() {  // area covered by a rectangle
    return (p1.x-p2.x)*(p1.y-p2.y);
  }
};
```

Initializing objects with different constructors

```
int main(int argc, char** argv) {
  Point p1(3,4), p2(15,9); // initialize points
  Rectangle r1(3,4,15,9);  // constructor 1 is called
  Rectangle r2(p1,p2);     // constructor 2 is called
  std::cout << r1.area() << std::endl; // prints 60
  std::cout << r2.area() << std::endl; // prints 60
  r1.p2.print();                       // prints (15,9)
  return 0;
}
```


Pointers to an object : objectPointers.cpp

```
#include <iostream>
#include <cmath>
class Point { ... }; // same as defined before
int main(int argc, char** argv) {
  // allocation to "stack" : p1 is alive within the function
  Point p1(3,4);
  // allocation to "heap" : *pp2 is alive until delete is called
  Point* pp2 = new Point(5,12);
  Point* pp3 = &p1; // pp3 is simply the address of p1 object
  p1.print();   // Member function access - prints (3,4)
  pp2->print(); // Member function access via pointer - prints (5,12)
  pp3->print(); // Member function access via pointer - prints (3,4)
  std::cout << "p1.x = " << p1.x << std::endl;         // prints 3
  std::cout << "pp2->x = " << pp2->x << std::endl;     // prints 5
  std::cout << "(*pp2).x = " << (*pp2).x << std::endl; // same to pp2->x
  delete pp2;  // allocated memory must be deleted
  return 0;
}
```


A quick UNIX tip : dos2unix

- If you create your source file in Windows and upload the source using WinSCP, sometimes you may encounter warnings from compiler, such as 'No newline at end of file'
- This happens due to slight difference in text file format in handling carriage return (Enter)
- `dos2unix [filename]` will convert the input file to UNIX format, so the warning should go away.

Using Standard Template Library (STL)

Why STL?

- Included in the C++ Standard Library
- Allows to use key data structure and I/O interface easily
- Objects behaves like built-in data types

Key classes

- Strings library : `<string>`
- Input/Output Handling : `<iostream>`, `<fstream>`, `<sstream>`
- Variable size array : `<vector>`
- Other containers : `<set>`, `<map>`, `<stack>`

STL in pratice

sortedEcho.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
int main(int argc, char** argv) {
  std::vector<std::string> vArgs; // vector of strings
  for(int i=1; i < argc; ++i) {
    vArgs.push_back(argv[i]);  // append each arguments to the vector
  }
  std::sort(vArgs.begin(),vArgs.end()); // sort the vector in alphanumeric order
  std::cout << "Sorted arguments :";    // print the sorted arguments
  for(int i=0; i < (int)vArgs.size(); ++i) { std::cout << " " << vArgs[i]; }
  std::cout << std::endl;
  return 0;
}
```

A running example

```
user@host:~/> ./sortedEcho Hi hello world 123 1 2
Sorted arguments : 1 123 2 Hi hello world
```


Using STL strings

```
int main(int argc, char** argv) {
  char* p = "Hello pointer";     // array of characters
  char* q = p; // q and p point to the same address
  p[0] = 'h';
  std::cout << p << std::endl; // "hello pointer"
  std::cout << q << std::endl; // "hello pointer"

  std::string s("Hello string"); // STL string
  std::string t = s;  // clones the entire string
  t[0] = 'h';
  std::cout << t << std::endl; // "hello string"
  std::cout << s << std::endl; // "Hello string" : s isn't changed

  // Below are possible with std::string, but not with char*
  s += ", you are flexible";   // s becomes "Hello string, you are flexible"
  t = s.substr(6,6);  // t becomes "string"
  return 0;
}
```


Using STL vectors

```
int main(int argc, char** argv) {
  int A[] = {3,6,8};  // the array size is fixed
  int* p = A;         // p and A points to the same array
  p[0] = 10;
  std::cout << ( A[0] == 3 ) << std::endl; // false, not any more

  std::vector<int> v; // vector is a variable-size array
  v.push_back(3);     // v contains 3
  v.push_back(6);     // v contains 3,6
  v.push_back(8);     // v contains 3,6,8
  std::vector<int> u = v;
  u[0] = 10;
  std::cout << ( v[0] == 3 ) << std::endl; // true
  return 0;
}
```


Algorithm InsertionSort

Data: An unsorted list $A[1 \cdots n]$
Result: The list $A[1 \cdots n]$ is sorted
for $j = 2$ **to** n **do**
  $key = A[j]$;
  $i = j - 1$;
  while $i > 0$ *and* $A[i] > key$ **do**
    $A[i+1] = A[i]$;
    $i = i - 1$;
  end
  $A[i+1] = key$;
end

insertionSort.cpp - User Interface

insertionSort.cpp - main() function

```
int main(int argc, char** argv) {
  std::vector<int> v; // contains array of unsorted/sorted values
  int tok;            // temporary value to take integer input
  // read a series of input values from keyboard
  while ( std::cin >> tok ) { v.push_back(tok); }
  std::cout << "Before sorting:";
  printArray(v);    // print the unsorted values
  insertionSort(v); // perform insertion sort
  std::cout << "After sorting:";
  printArray(v);    // print the sorted values
  return 0;
}
```

How to feed input values

- By keyword - type [input value]+[RET] per each input entry, and put Ctrl+D when finished

STL Use in InsertionSort Algorithm

insertionSort.cpp - printArray() function

```
// print each element of array to the standard output
void printArray(std::vector<int>& A) {
// call-by-reference to avoid copying large objects
  for(int i=0; i < (int)A.size(); ++i) {
    std::cout << " " << A[i];
  }
  std::cout << std::endl;
}
```


STL Use in InsertionSort Algorithm

insertionSort.cpp - insertionSort() function

```
// perform insertion sort on A
void insertionSort(std::vector<int>& A) { // call-by-reference
  for(int j=1; j < A.size(); ++j) { // 0-based index
    int key = A[j];  // key element to relocate
    int i = j-1;     // index to be relocated
    while( (i >= 0) && (A[i] > key) ) { // find position to relocate
      A[i+1] = A[i]; // shift elements
      --i;           // update index to be relocated
    }
    A[i+1] = key;    // relocate the key element
  }
}
```


Recursion

Short defintion of recursion

Recursion See "Recursion".

More complete defintion of recursion

Recursion If you still don't get it, see: "Recursion"

Key components of recursion

- A function that is part of its own definition
- Terminating condition (to avoid infinite recursion)

Example of recursion

Factorial

```
int factorial(int n) {
  if ( n == 0 )
    return 1;
  else
    return n * factorial(n-1); // tail recursion - can be transformed into loop
}
```

towerOfHanoi

```
void towerOfHanoi(int n, int s, int i, int d) { // n disks, from s to d via i
  if ( n > 0 ) {
    towerOfHanoi(n-1,s,d,i); // recursively move n-1 disks from s to i
    // Move n-th disk from s to d
    std::cout << "Disk " << n << " : " << s << " -> " << d << std::endl;
    towerOfHanoi(n-1,i,s,d); // recursively move n-1 disks from i to d
  } // this is multi-way recursion, which cannot be easily written in loop
}
```


Euclid's algorithm

Algorithm GCD

Data: Two integers a and b
Result: The greatest common divisor (GCD) between a and b
if *a divides b* **then**
  return *a*
else
  Find the largest integer t such that $at + r = b$;
  return $\text{GCD}(r,a)$
end

Function gcd()

```
int gcd (int a, int b) {
  if ( a == 0 )  return b; // equivalent to returning a when b % a == 0
  else           return gcd( b % a, a );
}
```


A running example of Euclid's algorithm

Function gcd()

```
int gcd (int a, int b) {
  if ( a == 0 )  return b; // equivalent to returning a when b % a == 0
  else           return gcd( b % a, a );
}
```

Evaluation of gcd(477,246)

```
gcd(477, 246)
    gcd(231, 246)
        gcd(15, 231)
            gcd(6, 15)
                gcd(3, 6)
                    gcd(0, 3)
gcd(477, 246) == 3
```


Divide-and-conquer algorithms

Solve a problem recursively, applying three steps at each level of recursion

- **Divide** the problem into a number of subproblems that are smaller instances of the same problem
- **Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
- **Combine** the solutions to subproblems into the solution for the original problem

Binary Search

```
// assuming a is sorted, return index of array containing the key,
// among a[start...end]. Return -1 if no key is found
int binarySearch(std::vector<int>& a, int key, int start, int end) {
  if ( start > end ) return -1; // search failed
  int mid = (start+end)/2;
  if ( key == a[mid] ) return mid; // terminate if match is found
  if ( key < a[mid] ) // divide the remaining problem into half
    return binarySearch(a, key, start, mid-1);
  else
    return binarySearch(a, key, mid+1, end);
}
```


Recursive Maximum

```
// find maximum within an a[start..end]
int findMax(std::vector<int>& a, int start, int end) {
  if ( start == end ) return a[start];  // conquer small problem directly
  else {
    int mid = (start+end)/2;
    int leftMax = findMax(a,start,mid);  // divide the problem into half
    int rightMax = findMax(a,mid+1,end);
    return ( leftMax > rightMax ? leftMax : rightMax ); // combine solutions
  }
}
```


Using STL's sort : `stdSort.cpp`

```
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
int main(int argc, char** argv) { // sorting software using std::sort
  int tok;
  std::vector<int> v;
  if ( argc > 1 ) { // if argument is given, read from file
    std::ifstream fin(argv[1]);
    while( fin >> tok ) { v.push_back(tok); }
    fin.close();
  }
  else { // read from standard input if no argument is specified
    while( std::cin >> tok ) { v.push_back(tok); }
  }
  std::sort(v.begin(), v.end()); // Sort using the algorithm in STL
  for(int i=0; i < v.size(); ++i) {
    std::cout << v[i] << std::endl; // print out the content
  }
  return 0;
}
```


Using insertionSort : `insertionSort.cpp`

```
#include <iostream>
#include <fstream>
#include <vector>
void insertionSort(std::vector<int>& v); // insertionSort as defined before
int main(int argc, char** argv) {
  int tok;
  std::vector<int> v;
  if ( argc > 1 ) {
    std::ifstream fin(argv[1]);
    while( fin >> tok ) { v.push_back(tok); }
    fin.close();
  }
  else {
    while( std::cin >> tok ) { v.push_back(tok); }
  }
  insertionSort(v); // differs from stdSort in only this part
  for(int i=0; i < v.size(); ++i) {
    std::cout << v[i] << std::endl;
  }
  return 0;
}
```

STL Use in InsertionSort Algorithm

insertionSort.cpp - insertionSort() function

```
// perform insertion sort on A
void insertionSort(std::vector<int>& A) { // call-by-reference
  for(int j=1; j < A.size(); ++j) { // 0-based index
    int key = A[j];  // key element to relocate
    int i = j-1;     // index to be relocated
    while( (i >= 0) && (A[i] > key) ) { // find position to relocate
      A[i+1] = A[i]; // shift elements
      --i;           // update index to be relocated
    }
    A[i+1] = key;    // relocate the key element
  }
}
```


Running time comparison

Running example with 200,000 elements

```
user@host:~$ time sh -c 'seq 1 200000 | ~hmkang/Public/bin/shuf | ./insertionSort >
/dev/null'
0:17.42 elapsed, 17.428 u, 0.017 s, cpu 100.0% ...
user@host:~$ time sh -c 'seq 1 200000 | ~hmkang/Public/bin/shuf | ./stdSort > /dev/null'
0:00.36 elapsed, 0.346 u, 0.042 s, cpu 105.5% ...
```

Why is the speed so different?

- If you didn't compile with -O option, the code is not optimized and it will be substantially slow (but the above example is compiled with -O)
- The time complexity of insertion sort is $\Theta(n^2)$, but the time complexity of STL's sorting algorithm is $\Theta(n \log n)$.

Next Lecture

- Merge Sort
- Quicksort
- Lower bound of comparison-based sorting algorithms
- Elementary data structures