

Biostatistics 615 - Statistical Computing

Lecture 6 More on C++ Basics

Jian Kang

September 24, 2015

Basics of Arrays and Strings

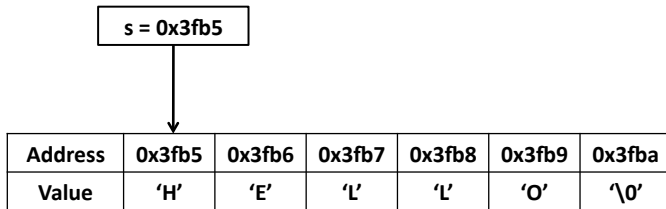
Array

```
int A[] = {3,6,8}; // A[] can be replaced with A[3]
std::cout << "A[0] = " << A[0] << std::endl; // prints 3
std::cout << "A[1] = " << A[1] << std::endl; // prints 6
std::cout << "A[2] = " << A[2] << std::endl; // prints 8
```

String as an array of characters

```
char s[] = "Hello, world"; // or equivalently, char* s = "Hello, world"
std::cout << "s[0] = " << s[0] << std::endl; // prints 'H'
std::cout << "s[5] = " << s[5] << std::endl; // prints ','
std::cout << "s = " << s << std::endl; // prints "Hello, world"
```

Pointers



```
char* s = "HELLO"; // array of {'H','E','L','L','O','\0'}
while ( *s != '\0' ) { // *s access the character value pointed by s
    std::cout << *s << std::endl; // prints 'H','E','L','L','O' at each line
    ++s; // advancing the pointer by one; points to the next element
}
```

Pointers and Loops

while loop

```
char* s = "HELLO"; // array of {'H','E','L','L','O','\0'}
while ( *s != '\0' ) {
    std::cout << *s << std::endl; // prints 'H','E','L','L','O' at each line
    ++s; // advancing the pointer by one
}
```

for loop

```
// initialize array within for loop
for(char* s = "HELLO"; *s != '\0'; ++s) {
    std::cout << *s << std::endl; // prints 'H','E','L','L','O' at each line
}
```

Pointers are tricky, but powerful

```
int A[] = {3,6,8}; // A is a pointer to a constant address
int* p = A;        // p and A are containing the same address
std::cout << p[0] << std::endl; // prints 3 because p[0] == A[0] == 3
std::cout << *p << std::endl;  // prints 3 because *p == p[0]
std::cout << p[2] << std::endl; // prints 8 because p[2] == A[2] == 8
std::cout << *(p+2) << std::endl; // prints 8 because *(p+2) == p[2]
int b = 3;        // regular integer value
int* q = &b;      // the value of q is the address of b
b = 4;            // the value of b is changed
std::cout << *q << std::endl;  // *q == b == 4

char s[] = "Hello";
char* t = s;
std::cout << t << std::endl; // prints "Hello"
char* u = &s[3]; // &s[3] is equivalent to s + 3
std::cout << u << std::endl; // prints "lo"
```

Pointers and References

```
int a = 2;
int& ra = a;    // reference to a
int* pa = &a;   // pointer to a
int b = a;      // copy of a
++a;           // increment a
std::cout << a << std::endl; // prints 3
std::cout << ra << std::endl; // prints 3
std::cout << *pa << std::endl; // prints 3
std::cout << b << std::endl; // prints 2
int* pb;        // valid, but what pb points to is undefined
int* pc = NULL; // valid, pc points to nothing
std::cout << *pc << std::endl; // Run-time error : pc cannot be dereferenced.
int& rb;        // invalid. reference must refer to something
int& rb = 2;    // invalid. reference must refer to a variable.
```

Functions

Core element of function

Type Type of return values

Arguments List of comma separated input arguments

Body Body of function with "return [value]" at the end

Functions

Core element of function

Type Type of return values

Arguments List of comma separated input arguments

Body Body of function with "return [value]" at the end

Defining functions

```
int square(int a) {  
    return (a*a);  
}
```


Functions

Core element of function

Type Type of return values

Arguments List of comma separated input arguments

Body Body of function with "return [value]" at the end

Defining functions

```
int square(int a) {  
    return (a*a);  
}
```

Calling functions

```
int x = 5;  
std::cout << square(x) << std::endl; // prints 25
```

Handling command line arguments

argConv.cpp - convert arguments in different format

```
#include <iostream>
#include <cstdlib> // needed for using atoi and atof functions
int main(int argc, char** argv) {
    for(int i=1; i < argc; ++i) {
        std::cout << argv[i] << "\t" << atoi(argv[i])
                    << "\t" << atof(argv[i]) << std::endl;
    }
    return 0;
}
```

Compiling and running argConv.cpp

```
user@host:~/ $ ./argConv 1 1.5 hello 2/3
```

1	1	1	
1.5	1	1.5	(atoi recognize only integer)
hello	0	0	(non-numeric portion will be recognized as zero)
2/3	2	2	(only recognize up to numeric portion)

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 2
    std::cout << x << std::endl;      // prints 1
    std::cout << bar(y) << std::endl; // prints 2
    std::cout << y << std::endl;      // prints 2
    return 0;
}
```

Call by value vs. Call by reference

Call-by-value is useful

- If you want to avoid unwanted changes in the caller's variables by the callee
- If you want to abstract the callee as a function only between inputs and outputs.

Call-by-reference is useful

- If you want to update the caller's variables by invoking the function.
- If you want to avoid copying an object consuming large memory to reduce memory consumption and computational time for copying the object.
 - As an extreme example, passing an 1GB object using call-by-value consumes additional 1GB of memory, but call-by-reference requires almost zero additional memory.

Let's implement Fisher's exact Test

	Treatment	Placebo	Total
Cured	a	b	a+b
Not cured	c	d	c+d
Total	a+c	b+d	n

Let's implement Fisher's exact Test

	Treatment	Placebo	Total
Cured	a	b	a+b
Not cured	c	d	c+d
Total	a+c	b+d	n

Desired Program Interface and Results

```
user@host:~/$ ./fishersExactTest 1 2 3 0
Two-sided p-value is 0.4
user@host:~/$ ./fishersExactTest 2 7 8 2
Two-sided p-value is 0.0230141
user@host:~/$ ./fishersExactTest 20 70 80 20
Two-sided p-value is 5.90393e-16
```

Fisher's Exact Test

Possible 2×2 tables

	Treatment	Placebo	Total
Cured	x	$a+b-x$	$a+b$
Not cured	$a+c-x$	$d-a+x$	$c+d$
Total	$a+c$	$b+d$	n

Hypergeometric distribution Given $a + b, c + d, a + c, b + d$ and $n = a + b + c + d$,

$$\Pr(x) = \frac{(a+b)!(c+d)!(a+c)!(b+d)!}{x!(a+b-x)!(a+c-x)!(d-a+x)!n!}$$

Fishers's Exact Test (2-sided)

$$p_{FET}(a, b, c, d) = \sum_x \Pr(x) I[\Pr(x) \leq \Pr(a)]$$

intFishersExactTest.cpp - main() function

```
#include <iostream>

double hypergeometricProb(int a, int b, int c, int d); // defined later

int main(int argc, char** argv) {
    // read input arguments
    int a = atoi(argv[1]), b = atoi(argv[2]);
    int c = atoi(argv[3]), d = atoi(argv[4]);
    int n = a + b + c + d;
    // find cutoff probability
    double pCutoff = hypergeometricProb(a,b,c,d);
    double pValue = 0;
    // sum over probability smaller than the cutoff
    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 p = hypergeometricProb(x,a+b-x,a+c-x,d-a+x);
            if ( p <= pCutoff ) pValue += p;
        }
    }
    std::cout << "Two-sided p-value is " << pValue << std::endl;
    return 0;
}
```


hypergeometricProb() function

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

double hypergeometricProb(int a, int b, int c, int d) {
    int num = fac(a+b) * fac(c+d) * fac(a+c) * fac(b+d);
    int den = fac(a) * fac(b) * fac(c) * fac(d) * fac(a+b+c+d);
    return (double)num/(double)den;
}
```

Running Examples

```
user@host:~/ $ ./intFishersExactTest 1 2 3 0
Two-sided p-value is 0.4 // correct
user@host:~/ $ ./intFishersExactTest 2 7 8 2
Two-sided p-value is 4.41018 // INCORRECT
```

Considering Precision Carefully

factorial.cpp

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

int main(int argc, char** argv) {
    int n = atoi(argv[1]);
    std::cout << n << "! = " << fac(n) << std::endl;
}
```

Running Examples

```
user@host:~/$ ./factorial 10
10! = 362880      // correct
user@host:~/$ ./factorial 12
12! = 479001600   // correct
user@host:~/$ ./factorial 13
13! = 1932053504  // INCORRECT
```

new hypergeometricProb() function

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

double hypergeometricProb(int a, int b, int c, int d) {
    double num = fac(a+b) * fac(c+d) * fac(a+c) * fac(b+d);
    double den = fac(a) * fac(b) * fac(c) * fac(d) * fac(a+b+c+d);
    return num/den; // use double to calculate factorials
}
```

Running Examples

```
user@host:~/ $ ./doubleFishersExactTest 2 7 8 2
```

Two-sided p-value is 0.023041

```
user@host:~/ $ ./doubleFishersExactTest 20 70 80 20
```

Two-sided p-value is 0 (fac(190) > 1e308 - beyond double precision)

How to perform Fisher's exact test with large values

Problem - Limited Precision

- `int` handles only up to `fac(12)`
- `double` handles only up to `fac(170)`

Solution - Calculate in logarithmic scale

$$\begin{aligned}\log \Pr(x) &= \log(a+b)! + \log(c+d)! + \log(a+c)! + \log(b+d)! - \log x! \\ &\quad - \log(a+b-x)! - \log(a+c-x)! - \log(d-a+x)! - \log n!\end{aligned}$$

$$\begin{aligned}\log(p_{FET}) &= \log \left[\sum_x \Pr(x) I(\Pr(x) \leq \Pr(a)) \right] \\ &= \log \Pr(a) + \log \left[\sum_x \exp(\log \Pr(x) - \log \Pr(a)) I(\log \Pr(x) \leq \log \Pr(a)) \right]\end{aligned}$$

logFishersExactTest.cpp - main() function

```
#include <iostream>
#include <cmath> // for calculating log() and exp()
double logHypergeometricProb(int a, int b, int c, int d); // defined later
int main(int argc, char** argv) {
    int a = atoi(argv[1]), b = atoi(argv[2]), c = atoi(argv[3]), d = atoi(argv[4]);
    int n = a + b + c + d;
    double logpCutoff = logHypergeometricProb(a,b,c,d);
    double pFraction = 0;
    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);
        }
    }
    double logpValue = logpCutoff + log(pFraction);
    std::cout << "Two-sided log10-p-value is " << logpValue/log(10.) << std::endl;
    std::cout << "Two-sided p-value is " << exp(logpValue) << std::endl;
    return 0;
}
```

Filling the rest

logHypergeometricProb()

```
double logFac(int n) {  
    double ret;  
    for(ret=0.; n > 0; --n) { ret += log((double)n); }  
    return ret;  
}  
  
double logHypergeometricProb(int a, int b, int c, int d) {  
    return logFac(a+b) + logFac(c+d) + logFac(a+c) + logFac(b+d) - logFac(a)  
        - logFac(b) - logFac(c) - logFac(d) - logFac(a+b+c+d);  
}
```

Running Examples

```
user@host:~/ $ ./logFishersExactTest 2 7 8 2  
Two-sided log10-p-value is -1.63801, p-value is 0.0230141  
user@host:~/ $ ./logFishersExactTest 20 70 80 20  
Two-sided log10-p-value is -15.2289, p-value is 5.90393e-16  
user@host:~/ $ ./logFishersExactTest 200 700 800 200  
Two-sided log10-p-value is -147.563, p-value is 2.73559e-148
```

Even faster

Computational speed for large dataset

```
time ./logFishersExactTest 1982 3018 2056 2944  
Two-sided log10-p-value is -0.863914, p-value is 0.1368  
0:10.17 elapsed ...
```

```
time ./fastFishersExactTest 1982 3018 2056 2944  
Two-sided log10-p-value is -0.863914, p-value is 0.1368  
0:00.00 elapsed,
```

Even faster

Computational speed for large dataset

```
time ./logFishersExactTest 1982 3018 2056 2944
Two-sided log10-p-value is -0.863914, p-value is 0.1368
0:10.17 elapsed ...
```

```
time ./fastFishersExactTest 1982 3018 2056 2944
Two-sided log10-p-value is -0.863914, p-value is 0.1368
0:00.00 elapsed,
```

How to make it faster?

- Most time consuming part is the repetitive computation of factorial
 - # of `logHypergeometricProbs` calls is $\leq a + b + c + d = n$
 - # of `logFac` call $\leq 9n$
 - # of `log` calls $\leq 9n^2$ - could be billions in the example above
- Key Idea is to store `logFac` values to avoid repetitive computation

newFac.cpp : new operator for dynamic memory allocation

```
#include <iostream>
#include <cstdlib>
int main(int argc, char** argv) {
    int n = atoi(argv[1]); // takes an integer argument
    double* facs = new double[n+1]; // allocate variable-sized array
    facs[0] = 1;
    for(int i=1; i <= n; ++i) {
        facs[i] = facs[i-1] * i; // calculate factorial
    }
    for(int i=n; i >= 0; --i) { // prints factorial values from n! to 0!
        std::cout << i << "! = " << facs[i] << std::endl;
    }
    delete [] facs; // if allocated by new[], must be freed by delete[]
    return 0;
}
```

Preambles and Function Declarations

```
#include <iostream>
#include <cmath>
#include <cstdlib>

// *** defined previously
double logHypergeometricProb(double* logFacs, int a, int b, int c, int d);

// *** New function ***
void initLogFacs(double* logFacs, int n);

int main(int argc, char** argv);
```

fastFishersExactTest.cpp - main() function

```
int main(int argc, char** argv) {
    int a = atoi(argv[1]), b = atoi(argv[2]), c = atoi(argv[3]), d = atoi(argv[4]);
    int n = a + b + c + d;
    double* logFacs = new double[n+1]; // *** dynamically allocate memory logFacs[0..n] ***
    initLogFacs(logFacs, n);           // *** initialize logFacs array ***
    double logpCutoff = logHypergeometricProb(logFacs,a,b,c,d); // *** logFacs added
    double pFraction = 0;
    for(int x=0; x <= n; ++x) {
        if ( a+b-x >= 0 && a+c-x >= 0 && d-a+x >=0 ) {
            double l = logHypergeometricProb(logFacs,x,a+b-x,a+c-x,d-a+x);
            if ( l <= logpCutoff ) pFraction += exp(l - logpCutoff);
        }
    }
    double logpValue = logpCutoff + log(pFraction);
    std::cout << "Two-sided log10-p-value is " << logpValue/log(10.) << std::endl;
    std::cout << "Two-sided p-value is " << exp(logpValue) << std::endl;
    delete [] logFacs;
    return 0;
}
```

fastFishersExactTest.cpp - other functions

```
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];  
}
```

Classes and user-defined data type

C++ Class

- A user-defined data type with
 - Member variables
 - Member functions

An example C++ Class

```
class Point { // definition of a class as a data type
public:      // making member variables/functions accessible outside the class
    double x; // member variable
    double y; // another member variable
};

Point p; // A class object as an instance of a data type
p.x = 3.; // assign values to member variables
p.y = 4.;
```

Adding member functions

```
#include <iostream>
#include <cmath>
class Point {
public:
    double x;
    double y;
    double distanceFromOrigin() { // member function
        return sqrt( x*x + y*y );
    }
};

int main(int argc, char** argv) {
    Point p;
    p.x = 3.;
    p.y = 4.;
    std::cout << p.distanceFromOrigin() << std::endl; // prints 5
    return 0;
}
```

Constructor - A better way to initialize an object

```
#include <iostream>
#include <cmath>
class Point {
public:
    double x;
    double y;
    Point(double px, double py) { // constructor defines here
        x = px;
        y = py;
    }
    // equivalent to -- Point(double px, double py) : x(px), y(py) {}
    double distanceFromOrigin() { return sqrt( x*x + y*y );}
};

int main(int argc, char** argv) {
    Point p(3,4) // calls constructor with two arguments
    std::cout << p.distanceFromOrigin() << std::endl; // prints 5
    return 0;
}
```

Constructors and more member functions

```
#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;
}
```


More class examples - pointRect.cpp

```
class Point { ... }; // same Point class as last slide
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
    // Passing user-defined data types by reference avoid the overhead of creating
    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;
}
```

Summary : Classes

- Class is an abstract data type
- A class object may contain member variables and functions
- Constructor is a special class for initializing a class object
 - There are also destructors, but not explained today
 - The concepts of default constructor and copy constructor are also skipped
- `new` and `delete` operators to dynamic allocate the memory in the heap space.