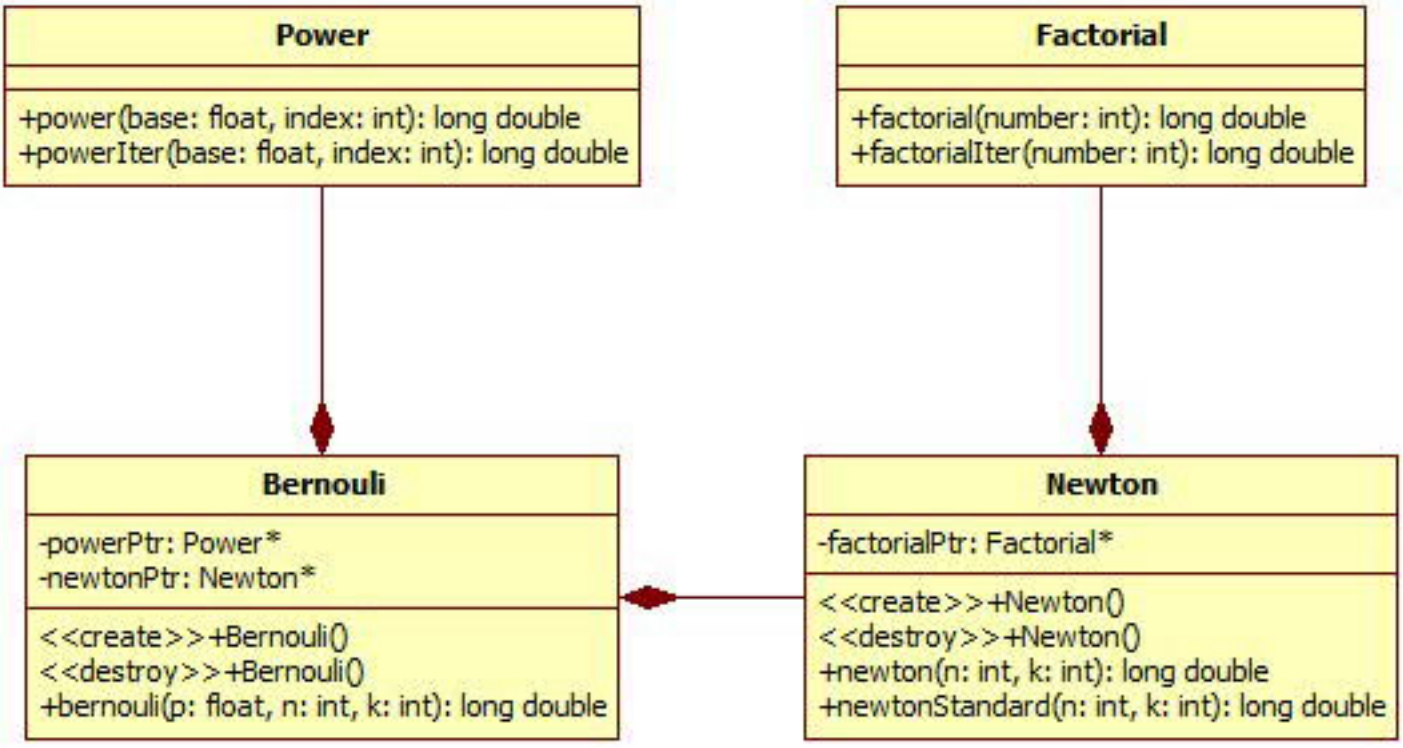
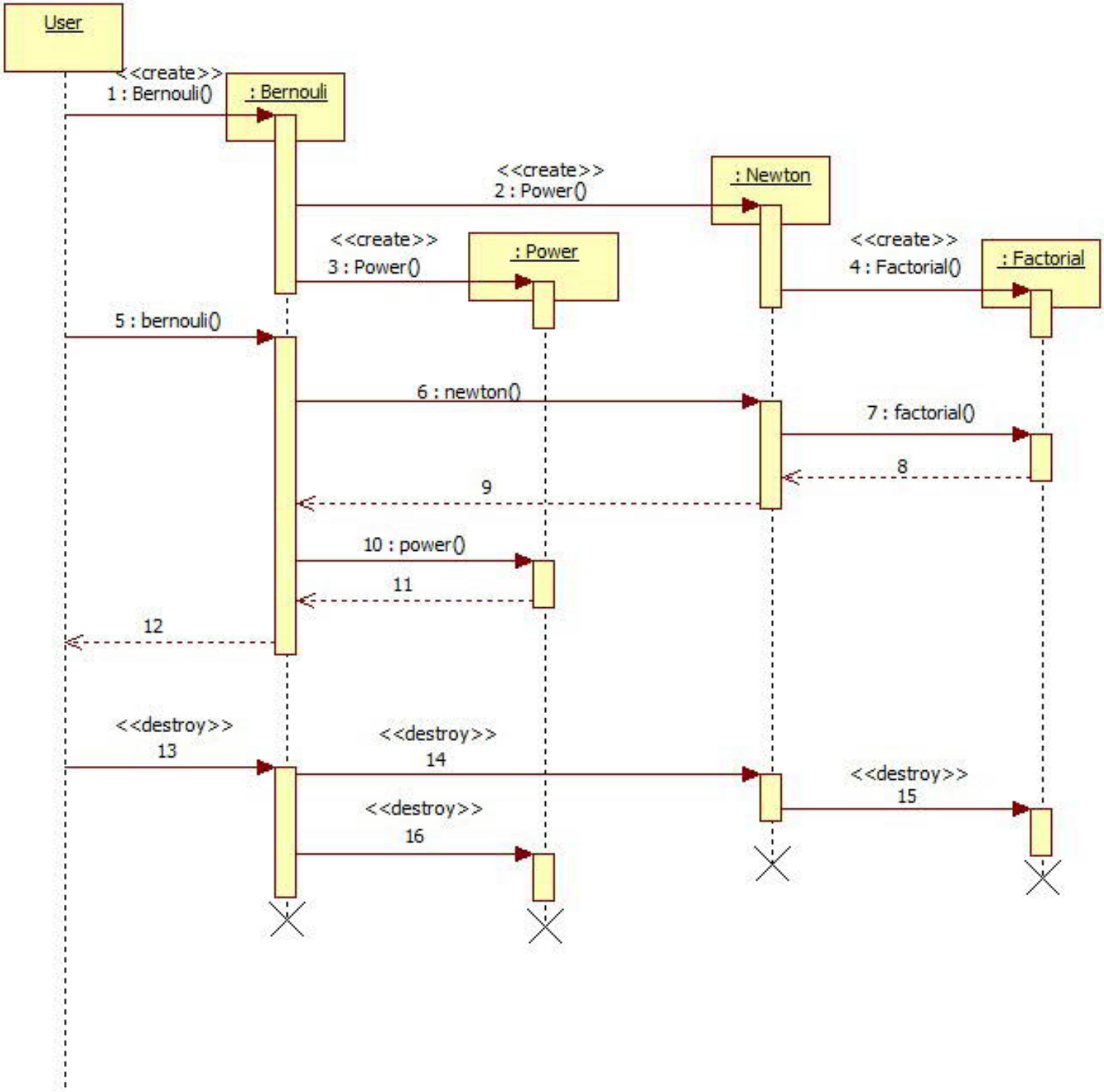
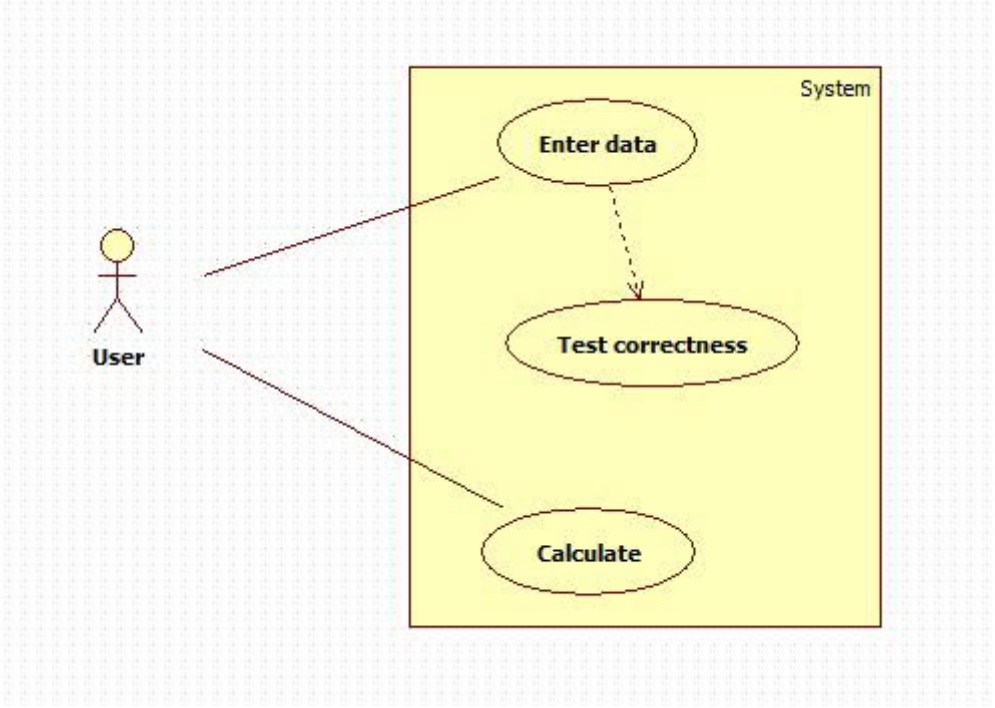

Power
+power(base: float, index: int): long double
+powerIter(base: float, index: int): long double
Factorial
+factorial(number: int): long double
+factorialIter(number: int): long double
Bernouli
-powerPtr: Power*
-newtonPtr: Newton*
<<create>>+Bernouli()
<<destroy>>+Bernouli()
+bernouli(p: float, n: int, k: int): long double
Newton
-factorialPtr: Factorial*
<<create>>+Newton()
<<destroy>>+Newton()
+newton(n: int, k: int): long double
+newtonStandard(n: int, k: int): long double


User
<<create>>
1 : Bernouli()
: Bernouli
<<create>>
2 : Power()
: Newton
<<create>>
3 : Power()
: Power
<<create>>
4 : Factorial()
: Factorial
5 : bernouli()
6 : newton()
7 : factorial()
8
9
10 : power()
11
12
<<destroy>>
13
<<destroy>>
14
<<destroy>>
15
<<destroy>>
16




```
#if !defined(_POWER_H)
#define _POWER_H

class Power {
public:
	long double power(float base, int index);
	long double powerIter(float base, int index);
};

#endif  //_POWER_H
//---------------------------------------------------------------------
#if !defined(_FACTORIAL_H)
#define _FACTORIAL_H

class Factorial {
public:
	long double factorial(int number);
	long double factorialIter(int number);
};

#endif  //_FACTORIAL_H
//---------------------------------------------------------------------
#include "Factorial.h"

#if !defined(_NEWTON_H)
#define _NEWTON_H

class Newton {
public:
	Newton();
	~Newton();
	long double newton(int n, int k);
	long double newtonStandard(int n, int k);
private:
	Factorial* factorialPtr;
};

#endif  //_NEWTON_H
//---------------------------------------------------------------------
```

```
#include "Power.h"
#include "Newton.h"

#if !defined(_BERNOULI_H)
#define _BERNOULI_H


class Bernouli {
public:
        Bernouli();
        ~Bernouli();
        long double bernouli(float p, int n, int k);
private:
        Power* powerPtr;
        Newton* newtonPtr;
};

#endif  //_BERNOULI_H
//---------------------------------------------

//---------------------------------------------
#include "Power.h"

long double Power::power(float base, int index) {
    if(index == 0) {
        return 1;
    }
    return base * power(base, index-1);
}

long double Power::powerIter(float base, int index) {
    long double result = 1;
    for(int i = 0 ; i < index; i++) {
        result *= base;
    }
    return result;
}
//-------------------------------------------------
#include "Factorial.h"

long double Factorial::factorial(int number) {
    if(number == 0) {
        return 1;
    }
    return number * factorial(number-1);
}

long double Factorial::factorialIter(int number) {
    long double result = 1;
    for(int i = 1; i <= number; i++) {
        result *= i;
    }
    return result;
}
```

```
//---------------------------------------------------------------------
#include "Newton.h"

Newton::Newton() {
    factorialPtr = new Factorial();
}

Newton::~Newton() {
    delete factorialPtr;
}

long double Newton::newton(int n, int k) {
    int N = n - k;
    int NbyK = 1;

    if(k >= N) {
        for(int i=k+1; i<=n; i++){
            NbyK *= i;
        }
        return NbyK/factorialPtr->factorial(N);
    } else {
        for(int i = N+1; i <= n; i++) {
            NbyK *= i;
        }
        return NbyK/factorialPtr->factorial(k);
    }
}

long double Newton::newtonStandard(int n, int k) {
    return factorialPtr->factorial(n) / (factorialPtr->factorial(k) * factorialPtr->factorial(n-k));
}
//---------------------------------------------------------------------
#include "Bernouli.h"

Bernouli::Bernouli() {
    powerPtr = new Power();
    newtonPtr = new Newton();
}

Bernouli::~Bernouli() {
    delete powerPtr;
    delete newtonPtr;
}

long double Bernouli::bernouli(float p, int n, int k) {
    float q = 1 - p;
    return newtonPtr->newton(n, k) * powerPtr->power(p, k) * powerPtr->power(q, n-k);
}
//---------------------------------------------------------------------------------------
```

```cpp
#include <iostream>
#include "Bernouli.h"

using namespace std;

bool isTheDataCorrect(float p, int n, int k) {
    return p <= 1 && p >= 0 && n >= 0 && k >= 0 && k <= n;
}

int main()
{
    float singleProbability;
    int numberOfAttempts, numberOfSuccesses;

    cout << "Enter the probability of a single success: ";
    cin >> singleProbability;
    cout << "Enter the number of attempts: ";
    cin >> numberOfAttempts;
    cout << "Enter the number of successes: ";
    cin >> numberOfSuccesses;
    if(isTheDataCorrect(singleProbability, numberOfAttempts, numberOfSuccesses)){
        Bernouli* calculator = new Bernouli();
        cout << "Probability of " << numberOfSuccesses << " successes in " << numberOfAttempts
        << " attempts is: " << calculator->bernouli(singleProbability,numberOfAttempts,
numberOfSuccesses) << "\n";
        delete calculator;
    } else {
        cout << "Error: Incorrect data\n";
    }
    return 0;
}
```

```
G:\IO\LB7\lab7_project\bin\Debug\lab7_project.exe
Enter the probability of a single success: 0.2
Enter the number of attempts: 10
Enter the number of successes: 3
Probability of 3 successes in 10 attempts is: 0.201327

Process returned 0 (0x0)   execution time : 9.817 s
Press any key to continue.
```