601.220 Intermediate Programming
FINAL REVIEW QUESTIONS
SOLUTIONS (answers bolded in blue)

These sample questions are meant to roughly convey the question formats and topic coverage of the final. This is not a sample final, in that it is not meant to be representative of the actual length of the final.

Code Tracing

Trace through each code fragment and write down the exact output that will be printed if the fragment is run. Code fragments may be assumed to be C++11, though many are equally valid in C99.

1.

```
int x = 0;
int y = -1;
bool a = false;
bool b = true;
if (a)
     cout << "a";
else
     cout << "b";
if (a || y)
     cout << "c";
cout << "d";
if ((x = 1) && (b = false)) {
     cout << "e";
}
cout << "\n" << x << " " << b;
```

Answer:
bcd
1 0

2.

```
int & alpha(int &l, int &r) {
    l += r;
    return r;
```

```
}
int main() {
    int x = 5;
    int y = 7;
    int &z = alpha(x, y);
    z += 3;
    cout << x << " " << y << " " << z;
    return 0;
}
```

Answer:
12 10 10

3.

```
int beta(int x) {
    cout << "i";
    return x + 5;
}
int beta(double x) {
    cout << "d";
    return x * 2;
}
int main() {
    int i = 5;
    double d = 5;
    cout << beta(i) << " ";
    cout << beta(i / 2.0) << " ";
    cout << beta(d) << " ";
    cout << beta(d / 2.0) << " ";
    return 0;
}
```

Answer:
i10 d5 d10 d5

4.

```
class Bar {
public:
    virtual void act() { cout << "Bar::act\n"; }
};

class Foo : public Bar {
public:
```

```
    virtual void act() { cout << "Foo::act\n"; }
};

int main() {
    vector<Bar> vec;
    vector<Bar*> vec2;
    vec.push_back(Foo());
    vec2.push_back(new Foo());
    vec[0].act();
    vec2[0]->act();
}
```

Answer:

```
Bar::act
Foo::act
```

5.

```
class Gamma {
public:
    virtual void one() { cout << "gamma-one "; }
    void two() { cout << "gamma-two "; }
};
class Delta : public Gamma {
public:
    virtual void one() { cout << "delta-one "; }
    void two() { cout << "delta-two "; }
};
int main() {
    Delta d;
    Gamma &c = d;
    d.one();
    d.two();
    c.one();
    c.two();
    return 0;
}
```

Answer:

```
delta-one delta-two delta-one gamma-two
```

6.

```
double zeta(double x, double y) {
```

```
   double &z = x;
   z += 5;
   return x + y + z;
}
double eta(double x, double y) {
   static double z = 10;
   z -= 5;
   return x + y + z;
}
int main() {
   double a = 3;
   double b = 7;
   cout << zeta(a, b) << " " << eta(a, b) << "\n";
   cout << zeta(b, a) << " " << eta(b, a) << "\n";
   return 0;
}
```

Answer:

```
23 15
27 10
```

7.

```
  std::vector<int> vec = { 1, 2, 3, 4, 5, 6, 7 };
  std::vector<int>::iterator iter;
  iter = vec.begin();
  *iter = 9;
  ++iter;
  ++iter;
  *iter = -4;
  iter = vec.end();
  iter--;
  *iter += 5;
  for (iter = vec.begin(); iter != vec.end(); iter++) {
      cout << *iter << " ";
  }
```

Answer:

```
9 2 -4 4 5 6 12
```

Code Writing

Write a code fragment to perform the specified task. It's a fragment, so you don't have to write `#include` statements or the header for `main()`, but you should otherwise write proper C++11 code that could be compiled and run correctly.

8.
Write a function that reads a list of numbers (from stdin), and returns the average (mean) value. The size of the list (i.e. how many numbers to read) is specified by the count parameter. If something goes wrong (use cin.fail() to check for this), throw a std::runtime error with an appropriate message. (You do not need to catch this exception.) The header is given to you below:

```
double averageNumbers(unsigned int count) {/* your code goes here */
}
```

Answer:

```
double averageNumbers(unsigned int count) {
      double sum = 0;
      for( unsigned int i = 0 ; i < count ; i++ ){
            double tmp;
            std::cin >> tmp;
            if( std::cin.fail() )
                throw std::runtime_error( "failure" );
            sum += tmp;
      }
      return sum / count;
}
```

9.
Fill in the code below for a C++ method named doubleSize() that takes as input a dynamically-allocated array of ints and an int capacity (passed as a reference) and which returns a pointer to a newly-allocated array with doubled capacity and the values from the old array (elements past the length of the old array may be left uninitialized). The function should also update the value of the capacity to match the new array's capacity.
Hint: You should take care to ensure this function does not leak memory. Also, be sure to use C++ style dynamic allocation (not C-style).

```
int * doubleSize(int * p, int & cap) {  /* your code goes here */ }
```

Answer:

```
int * doubleSize(int * p, int & cap) {
      int *newP = new int[cap * 2];
```

```
    for(int i = 0 ; i < cap ; i++)
        newP[i] = p[i];
    delete[] p;
    cap *= 2;
    return newP;
}
```

10.
Assume the following ListNode class exists:

```
class ListNode {
public:
    ListNode(int val, ListNode *nxt) : data(val), next(nxt) { }
    ~ListNode() { delete next; }
    int data;
    ListNode *next;
};
```

Further suppose that a linked list has been created out of ListNodes. Fill in the code below for a recursive function named printListRec which takes as input a ListNode * type variable which points to the head of an existing list, and outputs (to cout) the data values in the list starting at the head node, one per line. Solutions which do not make meaningful use of recursion will not earn full credit.

```
void printListRec(ListNode * head) { /* your code goes here */ }
```

Answer:

```
void printListRec(ListNode * head) {
  if (head) {
    cout << head->data << endl;  //output first item in list
    printListRec(head->next);    //output rest of list
  }
  //do nothing at all if head == nullptr
}
```

11.

```
class Bar {
public:
    Bar() : m_arr(new int[100]) {}
    ~Bar();
    int m_x;
```

```
    std::list<int> m_list;
    int* m_arr;
};

class Foo {
public:
    Foo() : m_barvec3(new vector<Bar>()) {}
    ~Foo();
    Bar m_bar;
    std::vector<Bar *> m_barvec1;
    std::vector<Bar> m_barvec2;
    std::vector<Bar> *m_barvec3;
};
```

Implement the destructors for Foo and Bar.

Answer:

```
~Bar(){ delete[] m_arr; }

~Foo() {
     for(size_t i=0; i<m_barvec1.size(); i++)
         delete m_barvec1[i];
     delete m_barvec3;
}
```

12.

```
class Foo {
     int _cSize;
     char* _cValues;
     std::vector< int > _iValues;
     double _dValues[100];
public:
     double& dValues( int i ){ return _dValues[i]; }
     int& iValues( int i ){ return _iValues[i]; }
     char& cValues( int i ){ return _cValues[i]; }
     int cSize( void ) const { return _cSize; }
     Foo( void ) : _cSize( 100 ) , _cValues( new char[_cSize] ){ }
};
```

Implement a proper copy constructor (deep copy) for Foo.

Answer:

```
Foo(const Foo& foo){
     _cSize = foo._cSize;
```

```
        _cValues = new char[_cSize];
        for(int i = 0 ; i < _cSize ; i++)
            _cValues[i] = foo._cValues[i];
        _iValues = foo._iValues;
        for(int i = 0 ; i < 100 ; i++)
            _dValues[i] = foo._dValues[i];
}
```

13.

Consider the following code.

```
//A class to represent a rational number, that is, a number that
//can be represented as the ratio of two integers
class Rational {
public:
  Rational (int num, unsigned int denom) : n(num), d(denom) {}
  std::string toString() const;

private:
  int n;
  unsigned int d;
};
```

Add to the class full definitions that:

a) allow two Rational numbers to be compared for equality via the == operator.
b) allow two Rational numbers to be added together using the + operator. The code should throw a std::range_error if either denominator is zero.
c) allow one Rational number to be incremented by another using the += operator. The code should throw a std::range_error if either denominator is zero.

(You do not need to show the method prototypes that would appear inside the class definition, and you don't need to catch the exceptions that your code might throw.)

Answer:

```
//Rational.cpp

#include <stdexcept>
#include "Rational.h"

bool Rational::operator==(const Rational& that) const {
  return this->n * that.d == that.n * this->d;
}

Rational Rational::operator+(const Rational& that) const {
  if (this->d == 0 || that.d == 0) {
```

```
        throw std::range_error("denominator of Rational was zero");
    }
    int new_denom = this->d * that.d;
    int new_num = this->n*that.d + that.n*this->d;
    return Rational(new_num, new_denom);
}

Rational& Rational::operator+=(const Rational& that){
    if (this->d == 0 || that.d == 0) {
        throw std::range_error("denominator of Rational was zero");
    }
    int new_denom = this->d * that.d;
    int new_num = this->n*that.d + that.n*this->d;
    this->n = new_num;
    this->d = new_denom;
    return *this;
}
```

Multiple Choice

14.

Which of these functions will NOT potentially cause a segmentation fault or compiler error/warning? Circle one choice.

A)

```
int * foo() {
    int x;
    return &x;
}
```

B)

```
int & foo() {
    int x;
    return x;
}
```

C)

```
std::string foo() {
    std::string s;
    return s;
}
```

D)

```
std::string & foo() {
    std::string s;
    return s;
}
```

Answer:

C

15.

Suppose a and b are two variables of type `int *`. Which statement changes the value in the memory location pointed to by a so that the value stored there matches the value pointed to by b?

A) `a = b;`
B) `&a = b;`
C) `a = &b;`
D) `*a = *b;`
Answer:
D

16.
What is the value of `a` at the end of this code segment?

```
int a = 7, b = 5;
int &c = a;
int &d = b;
c = d;
d = 8;
```

A) 5
B) 7
C) 8
D) this code won't compile
Answer:
A

17.
Consider the following C++ code, and in particular, Statements X and Y:

```
class Foo {
public:
    Foo(): x(0) { }
protected:
    int x;
};
class Bar : public Foo {
public:
    Bar(): Foo(), y(5) {
    x = 12; // Statement X
}
protected:
     int y;
```

```
};
int main() {
    Bar s = Bar();
    s.y = 24; // Statement Y
    return 0;
}
```

Which of the following is true about Statements X and Y, labeled above:
A) Statement X is legal, but Statement Y is not legal.
B) Statement X is not legal, but Statement Y is legal.
C) Both Statement X and Statement Y are legal.
D) Neither Statement X nor Statement Y is legal.
Answer:
A

18.
What is the output of the following program?

```
#include <iostream>
#include <exception>

using std::cout;

class One : public std::exception {};
class Two : public One {};

int main() {
    Two e;
    try {
        throw e;
        cout << "more stuff to do";
    }
    catch(Two &e) { cout << "Caught Exception Two "; }
    catch(One &e) { cout << "Caught Exception One "; }
    return 0;
}
```

A) Caught Exception Two more stuff to do
B) Caught Exception Two Caught Exception One
C) Caught Exception One
D) Caught Exception One more stuff to do
E) none of the above
Answer:

E (The actual output is Caught Exception Two.)