CS 7B - Spring 2014 - Review for Midterm 1. 3/11/14

1. Consider the following class definition for `Displacement`:

```cpp
using std::ostream;

class Displacement
{   friend Displacement operator*(double, Displacement);
    friend Displacement operator/(Displacement, double);
    friend ostream& operator<<(ostream& ostr, Displacement& x)  {
        ostr << x._m;
    }
  public:
    Displacement(double m=0) : _m(m) { }
    double m() const { return _m; }
    double cm() const { return 100*_m; }
    double km() const { return _m/1000; }
    double in() const { return _m/0.0254; }  // 1 in == 0.0254 m
    double ft() const { return _m/0.0254/12; }
    double mi() const { return _m/0.0254/12/5280; }
    void add_cm(double cm) { _m += cm/100; }
    void add_km(double km) { _m += 1000*km; }
    void add_in(double in) { _m += 0.0254*in; }
    void add_ft(double ft) { _m += 0.0254*12*ft; }
    void add_mi(double mi) { _m += 0.0254*12*5280*mi; }
    void subtract_cm(double cm) { _m -= cm/100; }
    void subtract_km(double km) { _m -= 1000*km; }
    void subtract_in(double in) { _m -= 0.0254*in; }
    void subtract_ft(double ft) { _m -= 0.0254*12*ft; }
    void subtract_mi(double mi) { _m -= 0.0254*12*5280*mi; }
    Displacement& operator=(const Displacement&);
    Displacement& operator*=(const double);
    Displacement& operator/=(const double);
  private:
    double _m;  // meters
};

Displacement operator*(double t, Displacement x)
{ Displacement y(t*x._m);
  return y;
}

Displacement operator/(Displacement x, double t)
{ Displacement y(x._m/t);
  return y;
}

Displacement& Displacement::operator=(const Displacement& x)
{ _m = x._m;
  return *this;
}

Displacement& Displacement::operator*=(const double t)
{ _m *= t;
  return *this;
}

Displacement& Displacement::operator/=(const double t)
{ _m /= t;
  return *this;
}
```

(a) Write out the definition of the constructor and describe how it works in detail.
Solution: `Displacement(double m=0) : _m(m) { }`
The constructor takes a single parameter, `m`, and uses it in the initialization list to assign that value to the private data member, `_m`. If no value is given then the default value of 0 is used to construct the `Displacement` object.

(b) Describe how this class overloads the assignment operator.
Solution: There are three assignment operator overloads whose prototypes are

```
Displacement& operator=(const Displacement&);
Displacement& operator*=(const double);
Displacement& operator/=(const double);
```

These are defined (presumably in the implementation) as

```
Displacement& Displacement::operator=(const Displacement& x)
{  _m = x._m;
   return *this;
}
```

This function takes a reference to a constant `Displacement` object as a parameter. That is the right hand operand in the assignment, whose private data member is then assigned to the private data member of the calling (implicit, left) operand, which is also a `Displacement` object. The calling object (`this`) is then dereferenced and returned as the output of the assignment operator function.

```
Displacement&
Displacement::operator*=(const double t)
{  _m *= t;
   return *this;
}
```

In this case the right operand is a multiplier. This can be an `int`, a `float`, or a `double` but it will be promoted to a `double` in performing the operation. The private data member of the calling (implicit) operand is multiplied by `t` and then the calling function's pointer (`this`) is dereferenced and returned.

```
Displacement& Displacement::operator/=(const double t)
{  m /= t;
   return *this;
}
```

Essentially the same thing as multiplication above except with division. Note that division by zero is not handled by an exception, as maybe it should be.

(c) Write a copy constructor for this class.
Solution: `Displacement(const Displacement& other) : _m(other._m) {}`

(d) What's wrong with using the following code as a copy constructor:

```
1 Displacement::Displacement( Displacement disp ) :
      _m( disp._m ) {}
```

Solution: With a copy constructor, you almost always want to pass by `const` reference.

(e) Write a driver for this class that will test each of the various member functions in a way that's easy to verify, or self-verifying. If might include output such as

```
x1 = 0 meters = 0cm.
x2 = 1 meters = 0.000621371mi.
x3 = 10 meters = 32.8084ft.
x4 = -5 meters = -196.85in.

x5 = 0.043496 miles + 39.3701 inches = 70.0254 meters.
```

etc.
Solution:

```
int main()
```

```
{
    int imult=20;
    double dmult = 2.1;
    // test constructor and insertion operator
    Displacement x(100);
    cout << "\nx=" << x << " meters";
    // test copy constructor
    Displacement w(x);
    cout << "\nw=" << w << " meters = " << x << " meters.";
    // test assignment operator;
    Displacement y = x;
    cout << "\ny=" << y << " meters = " << x << " meters.";
    Displacement z(11);
    // test assign a multiple
    z *= dmult;
    cout << "\nz=" << z << " meters = " << 2.1*11;
    // to be sure, it also works with an integer multiple:
    z *= imult;
    cout << "\nz=" << z << " meters = " << 2.1*11*20 << endl;
    // test conversions
    cout << z << " meters = " << z.cm() << " cm." << endl;
    cout << z << " meters = " << z.km() << " km."<< endl;
    cout << z << " meters = " << setprecision(15) << z.in() << " in = " << 39.37007874016
     *z.m() << endl;
    cout << z << " meters = " << z.ft() << " ft = " << 3.280839895013 *z.m() << endl;
    cout << z << " meters = " << z.mi() << " mi = " << 0.0006213711922373 *z.m() << endl;
    // test add functions
    double addx = double(rand()%100)/10.;
    cout << z << " meters + " << addx << " cm = ";
    z.add_cm(addx);
    cout << z << " meters."  << endl;
    cout << z << " meters + " << addx << " km = ";
    z.add_km(addx);
    cout << z << " meters." << endl;
    double temp = z.m();
    cout << setprecision(15) << z << " meters + " << addx << " in = ";
    z.add_in(addx);
    cout << z << " meters = " << 0.0254*addx+temp << endl;
    temp = z.m();
    cout << setprecision(15) << z << " meters + " << addx << " ft = ";
    z.add_ft(addx);
    cout << z << " meters = " << 0.3048*addx+temp << endl;
    temp = z.m();
    cout << setprecision(15) << z << " meters + " << addx << " mi = ";
    z.add_mi(addx);
    cout << z << " meters = " << 1609.344*addx+temp << endl;
     // test subtract functions
    addx = double(rand()%100)/10.;
    cout << z << " meters - " << addx << " cm = ";
    z.subtract_cm(addx);
    cout << z << " meters." << endl;
    cout << z << " meters - " << addx << " km = ";
    z.subtract_km(addx);
    cout << z << " meters." << endl;
    temp = z.m();
    cout << setprecision(15) << z << " meters - " << addx << " in = ";
    z.subtract_in(addx);
    cout << z << " meters = " << temp-0.0254*addx << endl;
    temp = z.m();
    cout << setprecision(15) << z << " meters - " << addx << " ft = ";
    z.subtract_ft(addx);
    cout << z << " meters = " << temp-0.3048*addx << endl;
    temp = z.m();
    cout << setprecision(15) << z << " meters - " << addx << " mi = ";
    z.subtract_mi(addx);
```

```
    cout << z << " meters = " << temp-1609.344*addx << endl;
    return 0;
}
```

That could be improved on, it involves a tad more human involvement than it maybe ought to do, but...here's a typical run:

```
x=100 meters
w=100 meters = 100 meters.
y=100 meters = 100 meters.
z=23.1 meters = 23.1
z=462 meters = 462
462 meters = 46200 cm.
462 meters = 0.462 km.
462 meters = 18188.9763779528 in = 18188.9763779539
462 meters = 1515.74803149606 ft = 1515.74803149601
462 meters = 0.287073490813648 mi = 0.287073490813633
462 meters + 4.1 cm = 462.041 meters.
462.041 meters + 4.1 km = 4562.041 meters.
4562.041 meters + 4.1 in = 4562.14514 meters = 4562.14514
4562.14514 meters + 4.1 ft = 4563.39482 meters = 4563.39482
4563.39482 meters + 4.1 mi = 11161.70522 meters = 11161.70522
11161.70522 meters - 6.7 cm = 11161.63822 meters.
11161.63822 meters - 6.7 km = 4461.63822 meters.
4461.63822 meters - 6.7 in = 4461.46804 meters = 4461.46804
4461.46804 meters - 6.7 ft = 4459.42588 meters = 4459.42588
4459.42588 meters - 6.7 mi = -6323.17892 meters = -6323.17892
```

2. Create a class called `PolarCoords` with two private data elements: `Displacement` and `Angle` and provide methods for

   (a) overloading the addition operator. Think of the sum as you would the sum of two 2-dimensional spatial vectors.

   Solution: First, here is code that fits the bill, and then some:

```
#define PI 3.14159265358979323846264338832795
using std::cout;
using std::istream;
using namespace std;

inline double abs(double x) {return x>=0 ? x : (-x);}
double tan(double);
double sin(double);
double cos(double);
double arctan(double);
double sqrt(double);

class PolarCoords {
    friend ostream& operator<<(ostream& ostr, const PolarCoords& x) {
        ostr << '(' << x._r << ", " << x._angle << ')';
        return ostr;
    }
    friend istream& operator>>(istream& istr, PolarCoords& pc)
    {
        char L;
        istr >> L >> pc._r >> L >> pc._angle >> L;
        return istr;
    }
public:
```

```
    PolarCoords(double r=0, double angle=0) : _r(r), _angle(angle) {}
26  enum Mode {POL, RECT};
    PolarCoords operator+(PolarCoords);
28  double get_r() {return _r;}
    double get_angle() {return _angle;}
30 private:
    double _r; // signed distance from origin
32  double _angle; // polar angle
    double _x; // horizontal coordinate
34  double _y; // vertical coordinate
    // private methods for setting values
36  void set_r();
    void set_angle();
38  void set_x();
    void set_y();
40 };

42 PolarCoords PolarCoords::operator+(PolarCoords x) {
    double xcoord1 = x._r*cos(x._angle);
44  double xcoord2 = _r*cos(_angle);
    double ycoord1 = x._r*sin(x._angle);
46  double ycoord2 = _r*sin(_angle);
    double xcoord = xcoord1 + xcoord2;
48  double ycoord = ycoord1 + ycoord2;
    double angle = arctan(ycoord/xcoord);
50  if(xcoord < 0) angle += PI;
    double r = sqrt(xcoord*xcoord+ycoord*ycoord);
52  PolarCoords sum(r,angle);
    return sum;
54 }

56 void PolarCoords::set_x() {
    _x = _r*cos(_angle);
58 }

60 void PolarCoords::set_y() {
    _y = _r*sin(_angle);
62 }

64 double tan(double x) {
    double eps = 1e-15;
66  double tiny = 1e-30;
    double a = x;
68  double f = tiny;
    double C = f;
70  double D = 0;
    double Delta = 1e20;
72  double b = 1.;
    D *= a; // zero
74  D += b;
    D = 1/D;
76  //if(D==0) D = tiny;
    C = f;
78  C = x/C;
    C += b;
80  //if(C==0) C = tiny;
    Delta = C*D;
82  f *= Delta;
    a = -x*x;
84  while(abs(1.-Delta)>eps) {
        //cout << "\nf = " << f;
86      b += 2.;
        D *= a;
88      D += b;
        D = 1/D;
```

```
        C = a/C;
        C += b;
        Delta = C*D;
        f *= Delta;
    }
    return f;
}

double cos(double x) {
    double t = tan(x/2);
    return (1-t*t)/(1+t*t);
}

double sin(double x) {
    double t = tan(x/2);
    return 2*t/(1+t*t);
}

double arctan(double x) {
    double eps = 1e-15;
    double tiny = 1e-30;
    double a = x;
    double f = tiny;
    double C = f;
    double D = 0;
    double Delta = 1e20;
    double b = 1., i = 1.;
    double xsqr = x*x;
    D *= a; // zero
    D += b;
    D = 1/D;
    //if(D==0) D = tiny;
    C = f;
    C = x/C;
    C += b;
    //if(C==0) C = tiny;
    Delta = C*D;
    f *= Delta;
    a = x*x;
    while(abs(1.-Delta)>eps) {
        //cout << "\nf = " << f;
        b += 2.;
        D *= a;
        D += b;
        D = 1/D;
        C = a/C;
        C += b;
        Delta = C*D;
        f *= Delta;
        ++i;
        a = i*i*xsqr;


    }
    return f;
}

double sqrt(double n) {
    /*Returns the square root of n. Note that the function */
  /*We are using n/2 itself as initial approximation */
  double x = n/2;
  double y = 1;
  double e = 1e-14; /* e decides the accuracy level*/
  while(abs(x - y) > e)
  {
```

```
        y = n/x;
        x = (x + y)/2;
        //cout << "\nx = " << x;
    }
    return x;
}
```

(b) overloading the insertion operator.

(c) overloading the extraction operator (assume the user enters polar coordinates in the form “(¡r¿,¡theta¿)”.
Solution: All these solutions are contained in the class definition above. Also, since I mentioned in class the extra challenge of doing all this without using the `cmath` library, I've included some *pretty good* math functions. In particular the `tan()` and the `arctan()` functions use a fairly mathematically sophisticated algorithm (Lentz's algorithm) that takes a series expansion and condenses it into a continued fraction method that is fast and robust.

3. Consider the code below defining a `class` named `Queue`.

```
class Queue{
public:
    Queue(int s=100);    // sets the default maximum number at 100
    ~Queue();
    char front() const;
    char back() const;
    bool is_empty() const;
    bool is_full() const;
    void enter(const char);
    char leave();
    void print();
    int size();
private:
    char* _a;     // the queue itself: a dynamic array of char
    int _max;     // the maximum number of elements on the queue
    int _count;   // the number of elements on the queue
};
```

(a) How would you modify this code to make it a template class which could be a Queue of any type of object?
Solution: In short, add template <class T> before the declaration of the class and then change input/output parameters for the various data and method members accordingly. Alternately, write class <typename T>.

```
template <class T>
class Queue{
public:
    Queue(int s=100);    // sets the default maximum number at 100
    ~Queue();
    T& front() const;
    T& back() const;
    bool is_empty() const;
    bool is_full() const;
    void enter(const T);
    T leave();
    void print();
    int size();
private:
    T* _a;     // the queue itself: a dynamic array of T
    int _max;     // the maximum number of elements on the queue
    int _count;   // the number of elements on the queue
};
```

(b) How would you define various functions?

Soltuion: If you were paying attention in lecture, you saw how we not only developed working definitions for these functions, but they were posted in the lecture notes on the calendar page. We defined the contructor like so:

```
Queue(int s=100) : _max(s), _count(0) {
        _a = new T[_max];
        assert(_a != 0);
    }    // sets the default maximum number at 100
```

The destructor: `Queue() {delete [] _a;}` The `front()` and all the other methods:

```
template <typename T>
T& Queue<T>::front() const {
    assert (_count > 0);
    return _a[_count-1];
}

template <typename T>
T& Queue<T>::back() const {
    assert (_count > 0);
    return _a[0];
}

template <typename T>
bool Queue<T>::is_empty() const {
    return (_count == 0);
}

template <typename T>
bool Queue<T>::is_full() const {
    return (_count == _max);
}

template <typename T>
void Queue<T>::enter(const T& x) {
    _count++;
    assert(_count < _max);
    for(int i = _count-2; i >= 0; --i)
        _a[i+1]=_a[i];
    _a[0] = x;
}

template <typename T>
T& Queue<T>::leave() {
    assert(_count > 0);
    return _a[--_count];
}

template <typename T>
void Queue<T>::print() {
    //char tmp;
    cout << "\ncount = " << _count << endl;
    int cnt = _count;
    //Queue temp(_count);
    //while(!((*this).is_empty())) { //copy the Queue to temp
    while(cnt>0) {
        (*this).enter((*this).leave());
        cout << (*this).back();
        cnt--;
```

```
    }
}

template <typename T>
int Queue<T>::size() {
    return _count;
}
```

(c) Why is the `front()` function defined as constant?
Solution: A class's member function should be declared constant if it will not change the private data elements of the class. The method `front()` is intended only to report on the contents of the front of the queue, not to change anything.

(d) Why does the `enter()` function take a constant parameter?
Solution: The `enter()` will not change the parameter passed to it, only enter it in the queue.

(e) Write a definition to implement the `print()` function for the `template class`, Queue, outside the interface.
Solution: See the end of the solution to part (b) above.

(f) Write a definition for a `friend` function overloading the insertion operator (`<<`). The code below overloads the insertion operator of the `Date` class, for example.
Soltuion Sometimes I ask questions because I'd really like to know the answer. This was one of those times, it seems.
http://bytes.com/topic/c/answers/670287-msvs2005-standard-insertion-operator-overload-lnk2028 First, you need to declare the `operator<<` function as an "unbound friend" by giving it an independent template, like so:

```
template<class Type>    //unbound friend
friend ostream& operator<<(ostream&, Queue<Type>&);
```

Then you can overload the insertion function like so:

```
template<class Type>
ostream& operator<<(ostream& ostr, Queue<Type>& x) {
    int cnt = x._count;
    while(cnt>0) {
        ostr << x.front() << ' ';
        x.enter(x.leave());
        //ostr << x.back() << ' ';
        cnt--;
    }
    return ostr;
}
```

To test it, I wrote a driver:

```
int main()
{
    Queue<int> qi(10);
    for(int i=0; i < 8;++i) {
        qi.enter(i*i); // create any old list
        cout << "\nback = " << qi.back();
    }
    cout << endl << qi; //overloaded insertion
    return 0;
}
```

whose output is as expected:

```
back = 0
back = 1
back = 4
back = 9
back = 16
back = 25
back = 36
back = 49
0 1 4 9 16 25 36 49
```

4. Write code for each of the following:

   (a) Declare a pointer `ppc` to a `PolarCoords`.
   Solution: `PolarCoord *ppc;` (That was easy!)

   (b) Create a `PolarCoords pc1` and have `ppc` point to `pc1`.
   Solution:
   ```
   PolarCoords pc1;
   ppc = &pc1;
   ```

   (c) Create a dynamic array of 3 `PolarCoords` and use pointer arithmetic to intialize, display each of them.
   Solution:

```
1     PolarCoords* dapc = new PolarCoords[3];
      for(int i = 0; i < 3; ++i) {
3         PolarCoords temp((rand()%100)/100.,(rand()%315)/100.);
          *dapc = temp;
5         cout << "\ndapc["<<i<<"] = " << (*dapc++);
      }
```

   Produces this:

```
dapc[0] = (0.9, 2.92)
dapc[1] = (0.23, 1.4)
dapc[2] = (0.13, 0.22)
```

   (d) Write code to delete the memory you allocated in part (c).
   Solution: `delete[] dapc;`

   (e) Suppose that `pdapc` is a pointer to the first of a dynamic array of `PolarCoords`. Write a while loop to add these them all, treating each polar coordinate pair as the terminal point of a vector whose original point is at the origin. Use pointer arithmetic in your loop.

```
    PolarCoords* pdapc = new PolarCoords[3];
2     PolarCoords* start = pdapc;
      for(int i = 0; i < 3; ++i) {
4         PolarCoords temp((rand()%100)/100.,(rand()%315)/100.);
          *pdapc = temp;
6         cout << "\npdapc["<<i<<"] = " << (*pdapc++);
      }
8     PolarCoords* end = pdapc;
      PolarCoords sum(0,0);
10    for(pdapc = start; pdapc != end; ++pdapc) {
        sum = sum + *pdapc;
12      cout << "\nsum = " << sum;
      }
14    cout << "\nsum = " << sum;
```

(f) Write code that creates a dangling pointer to a `PolarCoords`.

Solution: A dangling pointer is essentially a pointer which still exists, even though the object it pointed to no longer exists. These are especially nasty bugs because they seldom crash the program until long after they have been created, making the cause of the crash an enigma. C++ is an object-oriented programming language that does not rely on garbage collection, so it's easy to create dangling pointers. Here are a few of the many ways to create a dangling pointer.

- This code snippet shows the dynamic array of PolarCoords being created, initialized and deleted, and then the deleted content is print out

```
PolarCoords* pdapc = new PolarCoords[3];
for(int i = 0; i < 3; ++i) {
    PolarCoords temp((rand()%100)/100.,(rand()%315)/100.);
    *pdapc = temp;
    cout << "\npdapc["<<i<<"] = " << (*pdapc++);
}
delete[] pdapc;
for(int i = 0; i<3; ++i)
        cout << "\npdapc[" << i << "] = " << *pdapc++;
```

The output is here:

```
pdapc[0] = (0.78, 1.23)
pdapc[1] = (0.27, 0.92)
pdapc[2] = (0.62, 1.18)
```

...seemingly impervious to having been deleted! But this is dangerous. The memory is no longer allocated, it is thrown back on the heap, where it still happens to exist, and is available, but not guaranteed to be, especially if any activity were to occur in between deleting and accessing that memory.

- Here's another common way to dangle a pointer. Say you define a type a function like so:

```
typedef PolarCoords* PCP;

PCP newPC(PolarCoords x) {
    PolarCoords tmp(x); // copy constructor
    return &tmp;
}
```

The function takes a `PolarCoords`, uses a copy constructor to make a local copy of it called `tmp` and then returns a reference to `tmp`. The trouble is, when you use this, say, like so:

```
PolarCoords* ppc1;
PolarCoords pc4(1,6.28);
PCP pol1ptr = newPC(pc4);
cout << "\n*pol1ptr = " << *pol1ptr << endl;
```

...the output of the program is not what you might expect:

```
*pol1ptr = (2.51656e-307, 2.55196e-307)
```

The reason is that when you return from the function, the value of the `tmp` is deleted because it's gone out of scope. So...dangling pointer.

For more of these, see `http://www.ccs.neu.edu/home/will/CPP/dangling.html`

(g) Write code to dereference a pointer to a `PolarCoord`.

Solution: Too easy, huh? Ok, well, here it is anyway:

```
PolarCoords pc1(1,0);
PolarCoords* ppc = &pc1;
cout << *ppc;
```

That will print "`(1,0)`"

(h) Write code to display the address of a pointer to a `PolarCoord` on the console.
Simply append this line after the code from part (g) above: **cout ¡¡ ppc;**

(i) Given the following
`PolarCoords pcArray[5]`
`PolarCoord* pcArrayPtr = pcArray;`
How would you initialize the array to contain the five roots of unity on the unit circle (in polar form) and the use the pointer with pointer arithmetic in a loop to print these to the console?

Solution: In polar form, the roots of unity are simple: $(r, \theta) = \left(1, \frac{2\pi \cdot i}{n}\right), i = 0, 1, 2, \ldots, n-1$. So here's a function to do that:

```
void nthrootof1(PCP &ru, double n) {
    PolarCoords* start = ru;
    for(double i = 0; i<n; i++, ru++) {
        PolarCoords temp(1,2*i*PI/n);
        *ru = temp;
    }
    ru = start;
}
```

...you would call the function like so:

```
PolarCoords pcArray[5];
PolarCoords* pcArrayPtr = pcArray;
nthrootof1(pcArrayPtr, 5);
for(int i = 0; i < 5; ++i) cout << pcArrayPtr[i] << endl;
```

...which produces this output:

```
(1, 0)
(1, 1.25664)
(1, 2.51327)
(1, 3.76991)
(1, 5.02655)
```

(j) What type of `const` pointer is this and what permissions does the following expression have?:
`int * const vPtr = &x;`
This is a constant pointer to nonconstant data. Values from the address that the pointer points to can be modified, but not the pointer address (it will always point to same location). example: an array name is a `const` pointer
`*vPtr = 7; //legal`
`vPtr = &y //illegal, new address can't be assigned to a const pointer`

(k) What type of `const` pointer is this and what permissions does the following expression have?:
`const int *const vPtr = &x;`
Solution: It's a constant pointer to constant data. Neither the values that the pointer points to, nor its address can be modified.

(l) What's wrong with the following code?

```
PolarCoord* firstlast(PolarCoord pc[ ], int size) {
  PolarCoord pc1[2];
  pc1[0] = pc[0];
  pc1[1] = pc[1];
  return pc1;
}
```

Solution: This is a dangling pointer! Values will eventually be overwritten by other function calls.

5. Draw pictures to show the effects of the following statements:

```
string* str1 = new string(''Don Knuth");
string str2 = *str1; // str2 is a copy of *str1
string& str3 = *str1; // str3 is a synonym for *str1
string* str4 = &str2; // str4 points to str2
str3[5] = '?'; // originally mistakenly written str4[5] = '?'
str3.erase(3, 2); // originally mistakenly written str3->erase(3,2)
str2[1] = 'G';
str1->replace(2, 1,"$=$")
```

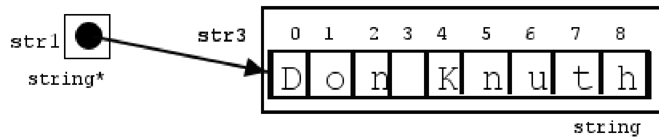
The original version of this had on lines 5 and 6, as indicated in the comments.
Running this code:

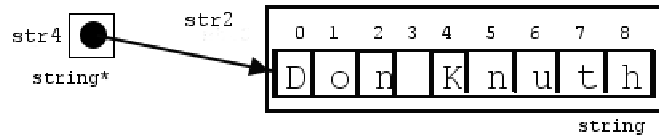
```

int main()
{
    string* str1 = new string("Don Knuth");
    cout << "\n*str1 = " << *str1;
    string str2 = *str1; // str2 is a copy of *str1
    cout << "\nstr2 = " << str2;
    string& str3 = *str1; // str3 is a synonym for *str1
    cout << "\nstr3 = " << str3;
    string* str4 = &str2; // str4 points to str2
    cout << "\n*str4 = " << *str4;
    str3[5]='?';
    cout << "\nstr3 = " << str3;
    str3.erase(3,2);
    cout << "\nstr3 = " << str3;
    str2[1] = 'G';
    cout << "\nstr2 = " << str2;
    cout << "\n*str4 = " << *str4;
    str1->replace(2,1,"$=$");
    cout << "\n*str1 = " << *str1;
}
```

...produces this output:

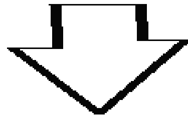
```
*str1 = Don Knuth
str2 = Don Knuth
str3 = Don Knuth
*str4 = Don Knuth
str3 = Don K?uth
str3 = Don?uth
str2 = DGn Knuth
*str4 = DGn Knuth
*str1 = Do$=$?uth
```

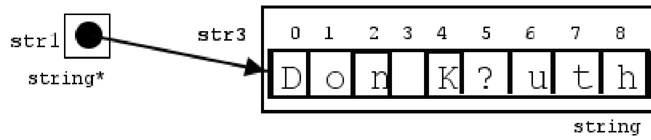
Which I illustrate (almost correctly) like so:

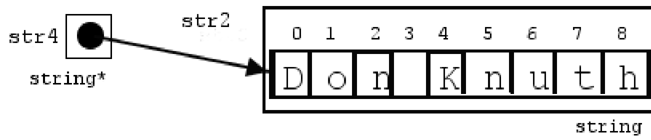




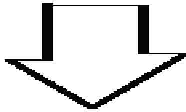

```
str3[5] = '?'
```

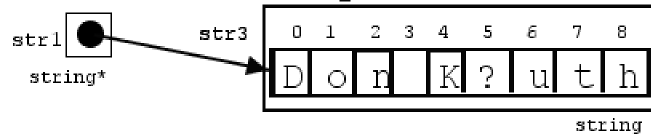


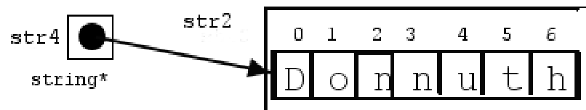


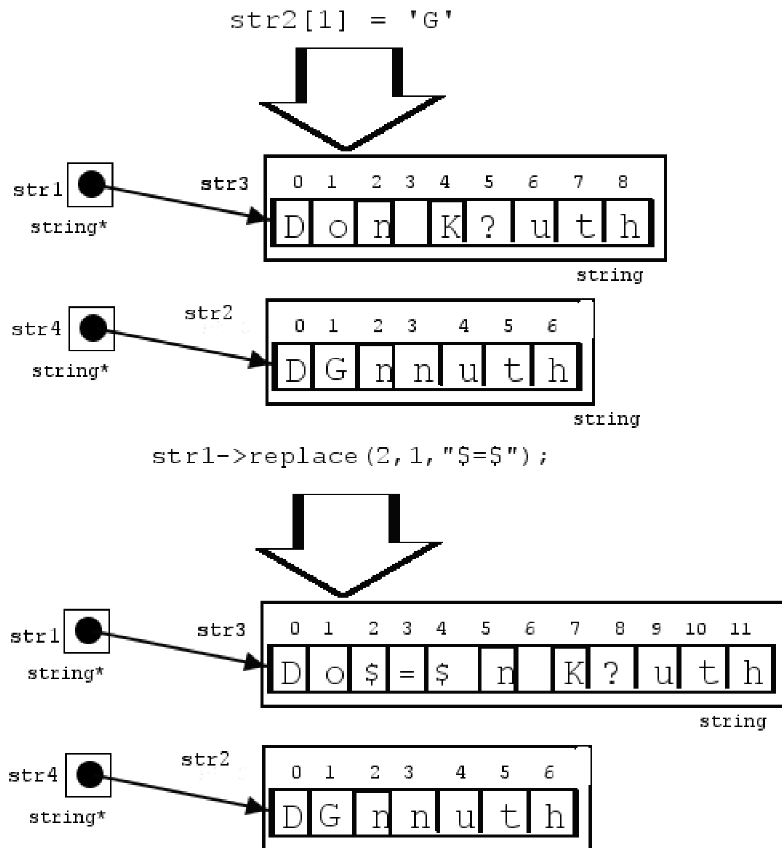



```
str4->erase(3,2)
```








str2[1] = 'G'
str1
string*
str3
0 1 2 3 4 5 6 7 8
D o n K ? u t h
string
str4
string*
str2
0 1 2 3 4 5 6
D G n n u t h
string
str1->replace(2,1,"$=$");
str1
string*
str3
0 1 2 3 4 5 6 7 8 9 10 11
D o $ = $ n K ? u t h
string
str4
string*
str2
0 1 2 3 4 5 6
D G n n u t h