```
George Owen --- CS32 Midterm Notes --- David Smallberg --- 4/25/2019

//use '\n', NOT '/n' - '/n' is out of date and can cause compiling errors

#include <iostream>
#include <cstdlib> //declares the exit() function, which causes the program to terminate
using namespace std; //all basic functions in c++ have the std:: header before them, this presumes that (ofc)

const double PI = 4 * atan(1.0);

class Circle
{
public:
    Circle(double x, double y, double r);
    void scale(double factor);
    void draw() const;
    double radius() const;
private:
        //Class invariant:
        //      m_r > 0
    double m_x;
    double m_y;
    double m_r;
}; //don't forget the semicolon!

double area(const Circle& x); //x is another name for the circle, and the circle will not change.
//saying Circle x and const Circle& x are the same in that they will not modify the thing being passed to the function. "Circle x" creates
// a copy, while "const Circle& x" passes the actual object, but promises not to modify it.

Circle::Circle(double x, double y, double r)
{
    if (r <= 0)
    {
        cerr << "Cannot create a circle with radius " << r << endl;
        exit(1);
    }
        m_x = x;
        m_y = y;
        m_r = r;
}
bool Circle::scale(double factor)
{
    if (factor <= 0)
        return false;
    m_r *= factor;
    return true;
}

double Circle::radius() const
{
    return m_r;
}

double area(const Circle& x)
{
    return PI * x.m_r * x.m_r;
}

int main()
{
    Circle blah(8, -3, 2.7)
    Circle c(-2, 5, 10);
    c.scale(2);
    c.draw();
    cout << area(c);
    cout << c.m_r;

    double x;
    cin >> x;
    if ( ! c.scale(x))
        exit(1);
}

=============================================
Creating a program with multiple source files
=============================================

Point.h //header filed contain class and function declarations
=========
class Point
{
    .....
};

Circle.h
=========
#include "Point.h"
class Circle
{
    ...
    Point m_center;
    double m_radius
};


myapp.cpp //cpp files contain function implementations. This one also contains the main routine
=========
#include "Circle.h"
#include "Point.h"

int main()
{
    Circle c;
    Point p;
}

//all the files included are linked together to make one executable
//tool that does this is called the LINKER
//benefit of splitting files:
    --easier to manage
    --CPP only needs to recompile files you're modifying. This means things compile way faster

class and struct == EXACTLY THE SAME THING IN C++
NOT the case in C#
    if you use struct, it defaults to starting with public members
    if you use class, it defaults to starting with private members

    class: generally used for more interesting things, as opposed to
    struct: generally used for a simple collection of data

---------------------------------------------
Student.h
=========

#ifndef STUDENT_INCLUDED //this is an include guard. it prevents your program from including the header file multiple times
#define STUDENT_INCLUDED    //-- if it hasn't been included (not defined), include and define it
#include "Course.h" //course.h also includes student.h -- this creates a CIRCULAR DEPENDENCY and breaks ya shit

class Student
```

```
{
    void enroll(Course* cp);
    ...
    Course* m_studylist[10]
};

#endif // STUDENT_INCLUDED

Course.h
=========
#ifndef COURSE_INCLUDED
#define COURSE_INCLUDED
//#include "Student.h" <- don't do this
class student; //instead, have an empty declaration at the start of the function. This way, the compiler knows student is a class it can create objects with, but doesn't cause a circular spiral of death

class Course
{
    ...
    Student* m_roster[100]
};

#endif // COURSE_INCLUDED

//If the file Foo.h defines the class Foo, when does another file require you to say
#include "Foo.h"
//and when can you instead simply provide the incomplete type declaration
class Foo;
//?

//You have to #include the header file defining a class whenever you:
    **Declare a data member of that class type
    **Declare a container (like a vector) of objects of that class type
    **Create an object of that class type
    **Use a member of that class type

class Blah
{
    ...
    void g(Foo f, Foo& fr, Foo* fp);  // just need to say   class Foo;
    ...
    Foo* m_fp;           // just need to say   class Foo; - because it's a pointer to the obj and not an actual obj
    Foo* m_fpa[10];      // just need to say   class Foo;
    vector<Foo*> m_fpv;  // just need to say   class Foo;
    Foo m_f;             // must #include Foo.h
    Foo m_fa[10];        // must #include Foo.h
    vector<Foo> m_fv;    // must #include Foo.h
};

void Blah::g(Foo f, Foo& fr, Foo* fp)
{
    Foo f2(10, 20);      // must #include Foo.h
    f.gleep();           // must #include Foo.h
    fr.gleep();          // must #include Foo.h
    fp->gleep();         // must #include Foo.h
}

=============================================
Steps of Constructing a Class/Struct Object:
=============================================

1. (not relevant yet)
2. Construct the data members, using the member initialization list; if a member is not listed these apply:
    * If a data member is a built-in type, it's left uninitialized
    * If a data member is of a class type, the default constructor is called for it
3. Execute the body of the constructor
```

```
struct Employee
{
    string name;
    double salary;
    int age;
};

Employee e; //Constructor for employee gets called when you run this

//there's no constructor for employee?
//if you declare no constructors for a class, the compiler writes a default(zero-argument) constructor for you.
//It looks like this:
Employee::Employee()
{}
======
class Circle
{
  public:
    Circle(double x, double y, double r);
      // no other Circle constructors are declared, so there's no default
      // constructor
    ...
  private:
    double m_x;
    double m_y;
    double m_r;
};

Circle::Circle(double x, double y, double r)
 : m_x(x), m_y(y), m_r(r)
{
    if (r <= 0)
    {
        ... write some error message ...
        exit(1);
    }
}

//Let's make a stick figure

class StickFigure
{
    public:
        StickFigure(double bl, double headDiameter, string nm, double hx, double hy);
        ...
    private:
    string m_name;
    Circle m_head;
    double m_bodyLength;
}

StickFigure::StickFigure(double bl, double headDiameter, string nm, double hx, double hy)
{
    if (bl <= 0)
    {
        cerr << "hes too smol" << endl;
        exit(1);
    }
    m_name = mn;
    m_head = Circle(hx, hy, headDiameter/2);
    m_bodyLength = bl;
}
```

data members are destroyed in the opposite order in which they're constructed

```
====================
Resource Management
====================
//let's make a string object
class String
{
    public:
        String(const char* value); //default constructor
        ~String(); //destructor
        String(const String& other); //copy constructor - called when creating a new string that's a copy of an existing string
        String& operator=(const String& rhs); //assignmnet operator - called when setting a string equal to another
        ...
    private:
        //class invariant
        //  m_text points to a dynamically allocated array of m_len+1 chars
        //  m_len > 0
        //  m_text[m_len] == '\0'
        char* m_text;
        int m_len;
}
//All strings have a pointer to a dynamically allocated array

String::String(const char* value) //default constructor
{
    if (value == nullptr)
        value = "";
    m_len = strlen(value);
    m_text = value;
    strcpy(m_text, value);
}

String::~String() //destructor
{
    delete [] m_text;
}

String::String(const String& other) //copy constructor
{
    m_len = other.m_len;
    m_text = new char[m_len+1];
    strcpy(m_text, other.m_text);
}

String String::operator=(const String& rhs) //assignment operator
{
    delete [] m_text;
    m_len = rhs.m_len;
    m_text new char[m_len+1];
    strcpy(m_text, rhs.m_text);
    return *this;
}

//don't have to give a function all of its arguments every time: if you don't want to, you can assign default values to them in the function declaration

void mwah(int a, int b = 42, int c = 20)
mwah(10, 20, 30);
mwah(10, 39); //c is 20
mwah(10); //b is 42 and c is 20
//once you assign a default value to a parameter, all parameters afterwards have to have one as well! how would you call the function w/ them otherwise? (you can't)


//if you allocate a single object with new, you must use the single form of delete
    p = new blah;
    delete p;
//if you allocate an array of objects instead, you have to use the array form
```

```
    p = new blah[10];
    delete [] p;

//initialization != assignment

//initialization (copy constructor is called)
    string s("Hello");
    string s2(s);
    string s3 = s //this is the COPY CONSTRUCTOR
//assignment (assignment operator is called)
    s2 = s;

//RAII: /resource acquisition is initialization

====================
    Linked Lists
====================

//4 types of data structures we've learned so far
    :/Fixed-Size Array
    :/Dynamically Allocated Array
    :/Resizeable array
    :/Linked List

Arrays: data structure w/ a collection of similar type data element
Linked Lists: data structure w/ a collection of unordered linked elements, aka nodes

//ADVANTAGES OF LINKED LISTS
    Linked Lists make it much easier to insert things into arrays/lists. Arrays have to shift all the objects down/deal with them in some way, but linked lists ya kinda just stick em in:
        -add an item
        -adjust some pointers
    Removing things from a linked lists:
        -adjust the pointers
        -delete the node

//DISADVANTAGE OF LINKED LISTS:
    Don't have immediate access to an arbitrary element of the list. The only way to get to an item is to follow the chain of pointers.


struct Node //nodes form the building blocks of linked lists
{
    int data; //values of the list are stored in this->data
    Node* next; //the definition for Node contains a pointer to a node
}
Node* head; //the head of the linked list. This can act as a dummy node or the actual first element

//Linked List Advice
    -draw pictures!
    -set a node's pointer members before changing other pointers
    -order matters
    -any time you write p->, make sure:
        --p has previously been given a value
        --p is not nullptr

class LinkedList //a sample linked list
{
public:
    LinkedList();
    void addToFront(string v);
    void addToRear(string v);
    void deleteItem(string v);
    bool findItem(string v);
    void printItems();
    ~LinkedList();
private:
    Node* head;
    struct Node //nodes can act as a private member struct
    {
        string value;
        Node *next;
        Node *prev;
    }
}

//allocating new nodes
    Node *p = new Node;
    Node *q = new Node;

//change/access node p's value
    p->value = "blah";
    cout << p->value;

//make p link to another node at address q
    p->next = q;

//get the address of the node after p
    Node *r = p->next

//make node q a terminal node
    q->next = nullptr;

delete p;
delete q;

void deleteItem(string v) //function to delete an arbitrary element of a singly linked list
{
    Node *p = head;
    while (p != nullptr)
    {
        if (p->next != nullptr && p->next->value == v)
            break; //if you find the value, break - p points to the node above

        p = p->next; //don't find the value, go down one
    }
    if (p != nullptr) //when you find the value, delete it
    {
        Node *killMe = p->next;
        p->next = killMe->next;
        delete killMe;
    }
}

bool search(Node* head, string v) //function to find a string in a singly linked list
{
    for (node *p = head; p != nullptr; p = p->next)
    {
        if (p->value == v)
            return true;
    }
    return false;
}

:/Doubly linked list:
    //the next + previous pointers contained in each nodes
    //this->next points to the next item, this->prev points to the previous item
:/Circular doubly linked list
    //same as a DLL, but the last node in the array points to the first one.

:/Dummy Node == the first node
    //value isn't part of the list, and it's not initialized
    //first item of the list is at head->m_next;, last one's at head->m_prev

int cmpr(Node* head, int* arr, int arr_size) //function to compare an array and linked
```

```
list and return the number of consecutive elements they share
{
    Node* p = head->next;
    for (int sim = 0; sim < arr_size; sim++)
    {
        if ((p->value == nullptr) || (arr[sim] != p->value))
            break;
        p = p->next;
    }
    return sim - 1;
}


//Places to check behavior:
    typical situation (activity in middle)
    at the head
    at the tail
    empty list
    1-element list
```