Section Handout 3

This handout was written by Derek Poppink and Hiroshi Ishii.

Overloading Constructors

Although we haven't finished learning about constructors in lecture, many of you would like to use them on your first assignment, so let's cover a few more aspects of them in section. Let's imagine a `Time` class.

```
class Time {
private:
      int ourtime;            // keeps track of minutes since midnight
      static int MaxHour;
      static int MaxMinute;   // we set these in the implementation file

public:
      Time(int hour, int minute, bool isMorning);
      int getHour() const;
      int getMinute() const;
      int getIsMorning() const;
};
```

The basic idea of the `Time` class is simple. We create a `Time` object by setting the hour and minute, and by indicating whether it's am or pm. The `Time` object can also return these values to us at any time.

You'll notice that we've chosen to represent the time internally with a single integer, tracking the number of minutes since midnight. This value will range from 0 to 1439. We could have instead stored hour, minute, and ismorning seperately. The choice of internal representation is yours, generally, and doesn't necessarily match the characteristics used outside the object.

```
Time::Time(int hour, int minute, boolean isMorning)
{
   ourtime = ((hour % MaxHour) * MaxMinute) + (minute % MaxMinute);
   if (!isMorning) ourtime += (MaxHour * MaxMinute);
}
```

Our constructor takes all three arguments, reduces them to the appropriate range, and transforms them to our internal representation.

But what if we want to create Time objects in other ways. Like other functions, methods and constructors can use overloading. For example, let's write a constructor which accepts military time:

```
Time::Time(int milhour, int minute)
{
   ourtime = ((milhour % (MaxHour * 2)) * MaxMinute) + (minute % MaxMinute);
}
```

Or we could define constructors which read the time from a string or file:

```
Time::Time(char *thetime);
Time::Time(ifstream &stream);
```

As mentioned in lecture, we are also able to use default arguments in constructors. For example, our first constructor could have a default value for `isMorning`. Be careful of using both default arguments and overloading, however. If we had also implemented the second constructor, the compiler would not be very accepting. It realizes that it would have no way of knowing which version was intended if a call was made like this:

```
Time *classtime = new Time(11, 0);
```

We also mentioned in lecture that compilers will sometimes create default constructors (constructors which take no arguments) for our classes. Using our `Time` class as an example, imagine we removed the existing constructor. If we then created Time objects statically or dynamically, memory will be allocated for all the instance variables, but they will not be initialized in any way. We have no guarantees about what `getHour()` or `getMinute()` would return, if called. Default constructors may still be useful, however, especially if we have other methods for use in setting the variables.

A reminder about default constructors: once you as the programmer declares any kind of constructor, the compiler will no longer generate a default constructor for your class. You are free to create your own default constructor, or provide default values for all the parameters in one of your existing constructors. This is useful when declaring an array of objects.

Private Methods

The classes we have looked at so far in lecture and in the handouts have maintained an interesting boundary between data members and methods. All the data members have been declared `private`, and all of the methods were marked as `public`. For the most part, this is a convenient distinction to make. Data members should never be public, as each object should have responsibility for it's own data. Methods usually exist for the class to interact with other parts of a program, and thus they are usually `public`. However, sometimes you will write `private` methods as well.

Under what circumstances are `private` methods a prudent plan? I'm glad you asked. One advantage of `private` methods is good decomposition. Consider an `Event` class, which has a name, a start time, and an end time.

```
class Event {
private:
      char *name;
      Time *start;
      Time *end;

public:
      Event(char *eventname, char *starttime, char *endtime);
      Time *getStart() const;
      Time *getEnd() const;
      …
}
```

We want to create `Event` classes from three strings passed to the constructor (perhaps we read them from a file), but we want the `Event` class to store the data as `Time` objects (taking advantage of the amazing features we've developed for that class).

Obviously, we're going to need to do some transformations to the strings in order to transform them into Time objects (detect colons, read am or pm, etc…). Rather than write ten nasty lines of code twice, create a `private` method of `Time` which transforms a single `string` to a `Time` object, and call it twice.

```
private:
      Time *stringToTime(char *time);
```

We don't want to declare the method as `public`, because we want clients to use the `Event` as it is, not tempt them with ways of circumventing our abstraction.

The same approach can be taken whenever you've got repeated code or verbose methods within a class. Sorting and allocating additional memory for data structures are other situations where `private` methods are likely to come up.

Cooperating Classes

No class is an island. Most C++ classes are meant to synchronize with other classes, like so many cogs in a machine. Just as the `Airport` and `Flight` classes in your assignment reference each other, so too will many other class groups you create.

As an example, consider a simple simulation of the new Trivial Pursuit: Star Wars Edition. The `Board` object keeps track of the question cards, the number of players, the game turns, and the `Players`. Each `Player` object must maintain their wedges, their position on the board, etc.... In order to move and answer questions, the `Player` object must also maintain a pointer to the `Board`.

```
class Board {
private:
      int numPlayers;
      Player *players;
      Cards *cards
public:
      Board();
      askQuestion();
}

class Player {
private:
   char *name;
   char *character;
   bool wedges[6];
   int position;
   Board *gameboard;
public:
   Player(char *name);
   void rollDie();
   void movePiece();
   void receiveWedge(enum color);
}
```

Interface for the Polygon Class

```
class Polygon {

  public:

  // default constructor
  Polygon(int n)
    : p_points(new Point[n]), p_n(n), p_max(n), p_area_valid(false) {}

  // initialize from array of Points
  Polygon(const Point points[], int n);

  Polygon(const Polygon &poly);           // copy constructor
  ~Polygon()                              // destructor
    { delete [] p_points; }               // OK even if p_points = 0


  int n() const { return p_n; }
  void n(int n);                          // sets the value of n

  Polygon & operator = (const Polygon &poly); // don't worry about this yet

  const Point & point(int i) const                // safe const accessor
    { assert(0<=i && i<=n()); return p_points[i]; }
  Point & point(int i);                          // unsafe non-const accessor

  void set(int i, const Point &p);
  void insert(int i, const Point &p);   // inserted point becomes elem i
  void append(const Point &p) { insert(n(), p); }
  void remove(int i);

  double area() const;

private:

  Point *p_points;          // array of points
  int p_n;                  // number of points
  int p_max;                // allocated storage

  int max() const { return p_max; }
  void max(int max);

  static void copy(Point p0[], const Point p1[], int n);

  mutable bool p_area_valid;    // is cached area valid?
  mutable double p_area;        // cached area of polygon

  double compute_area() const;  // computes area of polygon
};
```

Implementation of Polygon Class

```
Polygon::Polygon(const Point points[], int n)
  : p_points(new Point[n]), p_n(n), p_max(n), p_area_valid(false)
{
  copy(p_points, points, n);
}

// copy constructor
Polygon::Polygon(const Polygon &poly)
  : p_points(new Point[poly.p_n]), p_n(poly.p_n), p_max(poly.p_n),
    p_area_valid(poly.p_area_valid), p_area(poly.p_area)
{
  copy(p_points, poly.p_points, p_n);
}

Polygon& Polygon::operator=(const Polygon & poly)
{
  if(this != &poly) {
    if(poly.n() > max()) {
      Point *points_new = new Point[poly.max()];
      delete p_points;
      p_points = points_new;
      p_max = poly.max();
    }
    copy(p_points, poly.p_points, poly.p_n);
    p_n = poly.p_n;
    p_area_valid = poly.p_area_valid;
    p_area = poly.p_area;
  }
  return *this;
}

void Polygon::n(int n)
{
  if(n > max()) max(n);
  p_n = n;
}


// unsafe non-const accessor
Point& Polygon::point(int i)
{
  assert(0<=i && i<=n());
  p_area_valid = false;
  return p_points[i];
}

// this mutator is unsafe
void Polygon::set(int i, const Point &p)
{
  assert(0<=i && i<=n());
  p_area_valid = false;
  p_points[i] = p;
}
```

```
void Polygon::copy(Point p0[], const Point p1[], int n)
{
  for(int i=0; i<n; ++i)
    p0[i] = p1[i];
}

void Polygon::insert(int i, const Point &p)
{
  assert(0<=i && i<=n());
  if(n() + 1 > max())
  {
    Point *points_new = new Point[n() + 1];
    copy(points_new, p_points, n());
    delete p_points;
    p_points = points_new;
    p_max = n() + 1;
  }
  for(int j=n()-1; j>=i; --j)
    p_points[j+1] = p_points[j];
  p_points[i] = p;
  ++p_n;
  p_area_valid = false;
}

void Polygon::remove(int i)
{
  assert(0<=i && i<n());
  for(int j=i; j<n()-1; ++j)
    p_points[j] = p_points[j+1];
  --p_n;
  p_area_valid = false;
}

void Polygon::max(int max)
{
  assert(n() <= max);
  Point *points_new = new Point[max];
  copy(points_new, p_points, p_n);
  delete p_points;
  p_points = points_new;
  p_max = max;
}

double Polygon::area() const
{
  if(!p_area_valid) {
    p_area = compute_area();
    p_area_valid = true;
  }

  return p_area;
}
```

```
double Polygon::compute_area() const
{
  if(n() < 3)
    return 0.0;

  // only works for simple polygons

  double area = 0.0;

  for(int i=0; i<n()-1; ++i)
  {
    const Point &p0 = point(i);
    const Point &p1 = point(i+1);

    area += p0.x()*p1.y() - p0.y()*p1.x();
  }

  {
    const Point &p0 = point(n()-1);
    const Point &p1 = point(0);

    area += p0.x()*p1.y() - p0.y()*p1.x();
  }

  return 0.5*abs(area);
}
```