Intro to C++ classes and objects

- structs provide a way to group different data fields together into a single logical entity
- in ADTs we wanted to group together the data associated with an item and the operations on it
- structs (in their conventional use) don't give us that
- now we'll introduce classes, which allow us to group functions and data fields together into a single item
- classes also provide us with sophisticated mechanisms for designing data types that are built off of other data types, we'll introduce some of these shortly

C++ class

- we can define classes using a syntax much like we used for structs, but now we can also define functions within the class
  - subroutines are referred to as methods when they are part of a class, and as functions when they are standalone
- when we create variables based on our class they are referred to as *objects*, or as *instances* of that class
- we can make each part of a class private or private
  - private: only the class methods can access that item
  - public: any code can access it through an object of the class

C++ class syntax

- in the class definition we specify what is public and what is private, and give the prototypes for the methods
- all the class methods have access to all the class fields

```
class floatArray {
   private:  // we usually make the data fields private
      float* arr; // the pointer for the array
      int   allocated, inuse;  // how big an array did we create, how much of it is in use

   public: // we usually make the core methods public
      bool allocate(int size);  // allocate this much space for the array, return true iff ok
      bool set(int pos, float val); // try to set this value in this position, return true iff ok
      bool lookup(int pos, float &val); // try to lookup value in position, return true iff ok
};
```

Implementing the methods

- when we provide the full implementation of the methods we must specify which class (since different classes could have methods of the same name) and which method

```
bool floatArr::allocate(int size)
{
   allocated = 0; // floatArr methods have access to the private fields
   inuse = 0;
   arr = NULL;
   if (size > 0)  {
      arr = new float[size];
      if (arr != NULL) {
         allocated = size;
         return true;
       }
   }
   return false; // allocation failed due to bad size or insufficient memory
}
```

Creating objects, using methods

- we can create variables of the class type (aka objects/instances of the class) and call methods through the variable

```
int main()
{
   floatArr myArray;
   // try to allocate an array of size 10, then work with it if allocate succeeds
   if (myArray.allocate(10)) {
      for (int i = 0; i < 10; i++) {
           float val;
           cin >> val;
           if (myArray.set(i, val) {
              cout << “stored value “ << val << “ in position “ << i << endl;
           }
      }
   } // we should also have deallocated the array space!
}
```

Field/method access for “outside” code

- code outside the class can only directly access fields and methods that are public, not private

```
// suppose we have declared a class MyClass with
//     an int field, data, and a void method, foo
int main()
{
    MyClass somevar;

    somevar.foo();  // works iff foo was declared in the public section

    somevar.data = 10; // also works iff data was declared in the public section
}
```

Constructors and destructors

- there are special public methods associated with each class
- constructors are methods used to initialize the fields of a class, and a constructor is automatically run when an object is declared
  - the name of a constructor method is the same as the class name
- destructors methods “clean up” the class fields, automatically run when the object is destroyed (e.g. variable scope ends)
  - the name of a destructor is a ~ followed by the class name
- constructors and destructors have no return type

Constructor/destructor example

```
class Point {
   private:
      int x, y;
   public:
      Point(); // constructor
      ~Point(); // destructor
      void set(int xval, int yval);
      void print();
};

Point::Point()
{
   cout << “Enter x and y” << endl;
   cin >> x >> y;
}
```

```
Point::~Point()
{
    cout << “this is the end for “;
    print();
    cout << endl;
}

void Point::set(int xval, int yval)
{
   x = xval;
   y = yval;
}

void Point::print()
{
    cout << “(“ << x << “,” << y << “)”;
}
```

Example: when cons/dest run

```
int main()
{
   Point p;  // p's constructor automatically runs here
   p.set(5,10);
   p.print();
   return 0; // p's destructor automatically runs here
}
```

```
 int main()
 {
    Point a, b, c;  // constructors run for each

     // when main ends the destructors run on each
 }
```

Intro to inheritance

- classes can “inherit” fields and methods from other classes, then add their own (or replace inherited ones)

```
class Circle {
  protected:
     int x, y;
     float radius;
   public:
     Circle();
     ~Circle();
     void print();
     void setPt(int xval, int yval);
     void setRad(float rad);
};
```

```
class Sphere: public Circle {
    // automatically gets all Circle fields and methods
    // plus adds new fields and new methods
    protected:
        int z;
     public:
        Sphere();
        ~Sphere();
        void setZ(int zval);
        void print(); // override the inherited print
};
```

```
// ***protected: gives access to methods of classes that inherit from Circle
```

Specialization: inheritance heirarchies

- we can build a heirarchy in which classes get more and more specialized as they inherit: each new class adds just the extra fields/methods needed for its specialization