Intro to C++

Lecture Topics

- Objects, Constructors

These notes are taken from Prof. David Nicol.

Classes

- C++ grew out of C, pioneered by Bjarne Stroustrup at Bell Labs. It provides OBJECT ORIENTED programming model
  - Many features inherited from C, but then a number of new constructs; We will highlight some essential differences
- A `class` generalizes a `struct`
  - a `struct` gathers logically related data together
  - a `class` gathers logically related data together AND
  - allows to
    - control "who" can access that data
    - provide functions SPECIFIC TO THE CLASS
- Example: a vector has direction, `angle`, and magnitude, `length`.
- In C we might have the following structure

```
struct VectorStruct
{
     double angle;
     double length;
};

typedef struct VectorStruct vector;
```

  - and we may have functions that scale vectors, or add vectors

```
 void scaleVector(vector *v, double alpha)
 {
      v->angle *= alpha;
 }

vector addVectors(vector a, vector b)
{
      vector c;
      double ax = a.length*cos(a.angle);
      double bx = b.length*cos(b.angle);
      double ay = a.length*sin(a.angle);
      double by = b.length*sin(b.angle);
      double cx = ax+bx;
      double cy = ay+by;
      c.length = sqrt(cx*cx+cy*cy);
      c.angle = arcos( cx/c.length );

      return c;
}
```

- In C++ we could bundle this all together

```
#include <iostream>     // used for IO

using namespace std;    // creates textual container for
                        // variables and functions

class vector     // class is generalization of struct
{               // class constructors, do initialization
  public:      // public identifies the 'public exposure'
```

```
                // of this class to the 'outside' functions
           // that build instances, so-called 'constructors'
  vector(double a, double l) { angle = a; length = l; }
  vector() { angle =0.0; length = 0.0; }   // notice
                        // "overloading" of function name

  // functions tied to variables of this class
   double getAngle() { return angle; }
   double getLength() { return length; }
   void scale(double a) { length *= a; } // scale vector
   vector add(vector b);  // produce new vector by adding
                          // argument to variable
  protected: // compile throws error if direct access is
             // attempted
   double angle, length;
};

vector vector::add(vector b) { // produce new vector by
                             // adding argument to variable
      vector c;
      double ax = length*cos(angle);
      double bx = b.length*cos(b.angle);
      double ay = length*sin(angle);
      double by = b.length*sin(b.angle);
      double cx = ax+bx;
      double cy = ay+by;
      c.length = sqrt(cx*cx+cy*cy);
      c.angle = arcos( cx/c.length );

      return c;
}
```

- One declares a variable of type "vector" just as one would in C, using "vector" as the type, e.g.

```
vector b;
```

- But class variables are automatically initialized using "Constructor" functions declared in the class:

```
vector c(1.5, 2);
```

  - declares a class variable c, and initializes the angle to 1.5 and the length to 2.

- To call a class function, we use variable_name.function_name(function_args):

```
int main()
{
 vector c(1.5, 2);
 vector d(2.6,3);
 vector e = c.add(d);

 cout << "e length is " << e.getLength() << " and e angle
is " << e.getAngle() << "\n";
}
```

- Here `cout, cin, << , >>` are used for input and output, more on this later. For now, just know that the statement above prints out one line of text that gives the length

and angle of the vector variable e.

- Can also create and destroy dynamic class variables, e.g.

```
vector *aptr = new vector(1.5, 2.2);
vector *bptr = new vector(2.1, 5);
```

- When we're done with them, use "delete"

```
delete aptr;
delete bptr;
```

Class Specialization

- It is often the case that we want to specialize a class, perhaps add new functions that are used only for that class. C++ supports the notion of "derived classes" and "inheritance".
- Example: a special case of "vector" would be one that points straight up or down, or straight left or right. For these we might want another function that
  - computes the length of the hypotenuse of a triangle with two vectors as edges
  - retains all the functionality of a "vector" class
- When we declare the class, we declare that it "inherits" from "vector", and include only those parts that are different.

```
class orthovector: public vector
 {
      public:
       orthovector(int dir, double l) {  // dir is 0,1,2,3
                          \\ indicating right, up, left, down
         const double halfPI = 1.507963268;
         d = dir;
         angle = d*halfPI;
         length = l;
       }

       orthovector() { d = 0; angle = 0.0; length = 0.0; }

       double hypotenuse(orthovector b);

       protected:
         int d;
};

 double orthovector::hypotenuse(orthovector b) {
      if( (d+b.d)%2 == 0 ) return length+b.length;
      return (sqrt( length*length + b.length*b.length ) );
 }
```

- Note the following:
  - `orthovector: vector` declares inheritance, more on this later
  - new data item, "d" indicating direction (not entirely necessary but simple)
  - specialized constructors
  - member function that is available to orthovectors, but not to base vectors

Virtual functions

- A base class can be "incomplete" in the sense that it defines functions without bodies to them
  - these are called "virtual functions".

- derived classes declare instances of that function
- Example: a base class for colored regular geometric objects
  - base class gives specifics for color (common to all objects)
  - base class declares a virtual function "volume"
  - specialized classes (sphere, cube, pyramid) each define their own version of "volume"
- *More on this later*

Overloading Operators

- In C++ we can redefine operators like +,-,*, <, >, = to be whatever we want
  - think of them as functions of two variables that return a result.
  - This feature is very handy in programming objects, to capture common operations for example for vectors, addition (+):

```
class vector {

  // class constructors
   ...

  vector operator +(vector b)
  {
      vector c;
      double ax = length*cos(angle);
      double bx = b.length*cos(b.angle);
      double ay = length*sin(angle);
      double by = b.length*sin(b.angle);
      double cx = ax+bx;
      double cy = ay+by;
      c.length = sqrt(cx*cx+cy*cy);
      c.angle = arcos( cx/c.length );

      return c;
  }
  ...
};

 vector c(1.5, 2);
 vector d(2.6,3);

 vector e = c+d;
```

Some other C++ features

Strings

- C has the "idea" of a string, but C++ has a string class

```
 string s = "This is a string";
 string t = "This is another string";
 string u = s+t;  // concatenates strings s and t
```

Input/Output

- C has `stdin` and `stdout` as standard I/O streams
- C++ has same concepts, called `cin` and `cout`
  - C++ uses "<<" operator to direct formatted strings to a stream
  - C++ uses ">>" operator to direct input
  - Examples: output

```
cout << "Output sentence"; // prints string on screen,
                           // no end of line
cout << 120;   // prints 120 on screen, no end of line
cout << x;     // converts whatever form variable x has
               //to a string and prints
cout << "\n";  // end of line must be explicitly given
```

  - and input:

```
int age;
double weight;

cin >> age;   // take next string from cin, interpret as
              // an integer, store in variable age

cin >> weight; // take next string from cin, interpret as
               // a double, store in variable weight
```

  - `cin` grabs things one string at a time, but often want to get the whole line all at once
    - function `getline` does this

```
string mystr;

getline(cin,mystr); // loads mystr variable with line
                    // read from cin
```