Linked list implementation

What we're trying to create:

- suppose we want to maintain a list of circles in a 2d plane, each having x and y coordinates and a radius (all real numbers)
- we want to keep the circles in the order they were entered
- the list of circles could grow to any size
- we want to be able to add a circle
- we want to be able to print all the circles in a specific size range, e.g. all the circles with radius between 10 and 15, or between 0.5 and 1.6, etc

Chosen implementation approach

- we look at the requirements, and choose a linked list approach since
  - the number of circles could vary tremendously (not good for an array approach)
  - we **don't** need to find circles by their position in the list (which would have been slow in a linked list approach)

Identify needed data and functions

- our struct will need real numbers for x, y, and radius, plus a pointer for the next circle in the list
- our program will need to keep pointers for the first and last circles in the list
- we'll want functions to
  - create a new circle with given x,y,radius values
  - insert the new circle at the back of the list
  - search from the front of the list, printing all circles in a given radius (between passed rMin and rMax values)

Set data types and function profiles

- decide on the names and types for our circle struct

```
struct Circle { double x, y, radius; Circle *next; };
```

- decide on the names, parameter lists and return types for our functions

```
// allocate new circle with given stats, return pointer to it
Circle* create(double x, double y, double r);
// insert at back, update back ptr, return true iff successful
bool insert(Circle* &back, Circle *newcirc);
// search forward from front, printing all
// circles found with radius between minRand and maxRad
void search(Circle* front, double minRad, double maxRad);
```

Identify supporting functions/data

- we'll need some way to get commands from the user and either insert, search, or quit based on the command
  - possibly constants for the three command types
  - a function to get/return the user's next command

```
const char Quit = 'Q';
const char Insert = 'I';
const char Search = 'S';

// prompt the user and get their chosen command,
//    repeating until a valid command is obtained
// return the valid command
char getCommand();
```

Support functions continued

- we'll need a function to deallocate the list when done

```
void deallocate(Circle* &front);
```

- we'll need some way to get three numeric values from the user to pass to the create function
  - a function to get/return a positive number

```
// display the prompt and read the user's value
//    repeating until a positive number is provided
// return the final value
double getNumber(string prompt);
```

Implement incrementally

- implement one step at a time, compile and test after each
  - create skeletal versions of struct, functions, main
  - set up the main routine to use the functions
  - implement the processCommand routine
  - implement the getCommand routine
  - implement the getNumber routine
  - implement the create routine
  - implement the insert routine
  - implement the search routine
  - implement the deallocate routine

The definitions and prototypes

```
#include <iostream>
using namespace std;

struct Circle {
   double x, y, radius;
   Circle *next;
};

const char Quit = 'Q';
const char Insert = 'I';
const char Search = 'S';

Circle* create(double x, double y, double r);
bool insert(Circle* &front, Circle* &back, Circle *newcirc);
void search(Circle* front, double minRad, double maxRad);
char getCommand();
double getNumber(string prompt);
void deallocate(Circle* &front);

// main and the full function implementations will go below here
```

Skeletal main and functions

```
// initially just the bare minimum to get them to compile
int main() { }
Circle* create(double x, double y, double r) { return NULL; }
bool insert(Circle* &f, Circle* &b, Circle *newcirc) { return false; }
void search(Circle* front, double minRad, double maxRad) { }
char getCommand() { return Quit; }
double getNumber(string prompt) { return 0; }
void deallocate(Circle* &front) { }
```

Completing main

```
int main()
{
   Circle *front = NULL;
   Circle *back = NULL;
   char cmd;
   do {
      cmd = getCommand();
      // handle inserts
      if (cmd == Insert) {
         double x, y, r;
         x = getNumber("Enter x:");
         y = getNumber("Enter y:");
         r = getNumber("Enter radius:");
         Circle* tmp = create(x,y,r);
         if (tmp != NULL) {
            insert(front, back, tmp);
         }
      }
      // handle searches
      else if (cmd == Search) {
         double min, max;
         min = getNumber("Enter min radius:";
         max = getNumber("Enter max radius:";
         search(front, min, max);
      }
   } while (cmd != Quit);

   deallocate(front);
   return 0; // end of main
}
```

Completing getCommand

```
// typical prompt and read until they give a valid response
char getCommand()
{
   cout << “Enter “ << Insert << “ to insert,” << endl;
   cout << “   or “ << Search << “ to search,” << endl;
   cout << “   or “ << Quit << “ to quit,” << endl;
   char cmd;
   cin >> cmd;
   cmd = toupper(cmd);
   switch (cmd) {
      case Insert:
      case Quit:
      case Search:
         return cmd;
      default:
         cout << “That was an invalid command, “;
         cout << “please try again” << endl;
         return getCommand();
   }
}
```

Completing getNumber

```
// usual recursive get-a-number, fllushing buffer on garbage
double getNumber(string prompt)
{
   const int LineLen = 80; // max num input chars to clear
   cout << prompt << endl;
   double num;
   cin >> num;
   if (cin.fail()) {
      cin.clear();
      cin.ignore(LineLen, '\n');
      cout << “That was not a number, please try again” << endl;
      num = getNumber(prompt);
   }
   return num;
}
```

Completing create

```
Circle* create(double x, double y, double r)
{
   // create the new circle and make sure new worked
   Circle* newcirc = new Circle;
   if (newcirc != NULL) {
      // set all the field values
      newcirc->x = x;
      newcirc->y = y;
      newcirc->radius = r;
      newcirc->next = NULL;
   }
   // return the pointer to the “filled in” new circle
   return newcirc;
}
```

Completing insert

```
bool insert(Circle* &front, Circle* &back, Circle *newcirc)
{
   if (newcirc == NULL) {
      // we were given a non-existent circle to insert
      return false;
   } else if (front == NULL) {
      // this is the first and only item in the list so far,
      // so we need to update front and back to refer to it
      front = newcirc;
      back = newcirc;
      return true;
   } else {
      // this isn't the first item,
      // so we just need to update back
      back->next = newcirc; // old back item knows new one comes next
      back = newcirc; // back knows the new item is now the last
      return true;
   }
}
```

Completing search

```
void search(Circle* front, double minRad, double maxRad)
{
   // go from front of list to back, one item at a time
   //    NULL means we've hit end of list
   Circle* current = front;

   while (current != NULL) {
      // check the circle radius against the min/max we were given
      if ((current->radius >= minRad) && (current->radius <= maxRad)) {
         // found one!  print the current circle
         cout << "(" << curr->x << "," << curr->y << "):";
         cout << curr->radius << endl;
      }
   }
}
```

Completing deallocate

```
void deallocate(Circle* &front)
{
   // delete one item at a time until hit the end of list
   while (front != NULL) {
      // remember the one to be deleted
      Circle* victim = front;
      // advance front to point to the next one in line
      front = front->next;
      // deallocate the one to be deleted
      delete victim;
   }
}
```