```
-----------------------------------------------------------------
dlink_usage.cpp: double-linked list sorting using insertion sort
-----------------------------------------------------------------

#include <...>
using namespace std;

#include "person.h"

template <typename T>
class dlist {
  private:
    struct node {
      node() { key = T(); prev = next = this; }
      node(T &n_key) { key = n_key; prev = next = this; }

      T key;
      node *prev;
      node *next;
    };

    node *head;

  public:
    dlist() { head = new node(); }
    ~dlist() { ... }

    void push_back(T &);
    void insertion_sort();
    void mergesort();

    class iterator {
      public:
        iterator() { p = NULL; }

        T & operator*() { return p->key; }
        iterator operator++() { p = p->next; return *this; }
        bool operator!=(iterator &rhs) { return p != rhs.p; }

      private:
        iterator(node *np) { p = np; }
        node *p;

      friend class dlist<T>;
    };

    iterator begin() { return iterator(head->next); }
    iterator end() { return iterator(head); }
};
```

```
template <typename T>
void dlist<T>::push_back(T &key) {
  node *p = new node(key);
  node *pp = head->prev;

  p->next = head;
  p->prev = pp;

  pp->next = p;
  head->prev = p;
}

template <typename T>
void dlist<T>::insertion_sort() {
  node *p = head->next;
  node *pp, *pn, *q;

  while (p != head) {
    pp = p->prev;
    pn = p->next;

    // unlink node p
    pp->next = pn;
    pn->prev = pp;

    // find node q preceeding node p
    q = pp;
    while (q != head && p->key < q->key)
      q = q->prev;

    // relink node p
    p->next = q->next;
    q->next = p;
    p->prev = q;
    p->next->prev = p;

    p = pn;
  }
}

-----------------------------------------------------------------
Hint: Pointer p sweeps thru the double linked list from head to
tail. Pointer q uses a reverse order sweep to determine where to
place node p. Relinking takes care of (replaces) data movement.
-----------------------------------------------------------------
```

```
template <typename T>
struct dlist<T>::node *dlist<T>::merge(node *L, node *M, node *R) {
  node *H = L->prev;    // sublist head node
  node *p = L;          // node to consider from sublist L
  node *q = M;          // node to consider from sublist M

  // code intentionally left out -- see HW5

  return H->next;
}

template <typename T>
struct dlist<T>::node *dlist<T>::mergesort(node *L, node *R) {
  if (L->next == R)
    return L;

  int N = 0;

  node *p = L;
  while (p != R) {
    p = p->next;
    N++;
  }

  node *M = L;
  for (int i=0; i<N/2; i++)
    M = M->next;

  L = mergesort(L, M);
  M = mergesort(M, R);
  L = merge(L, M, R);

  return L;
}

template <typename T>
void dlist<T>::mergesort() {
  if (head->next == head->prev)
    return;

  head->next = mergesort(head->next, head);
}

#endif // DLIST
```

```
template <typename T>
void readdata(string &fname, dlist<T> &A) { ... }

template <typename T>
void printdata(T p1, T p2, string &fname) { ... }

int main(int argc, char *argv[]) {
  if (argc != 3) {
    cerr << "usage: " << argv[0]
         << " -insertion|mergesort file.txt\n";
    return 0;
  }

  string algname(&argv[1][1]);
  string fname_in(argv[2]);

  dlist<person_t> A;

  readdata(fname_in, A);


  if (algname.compare("insertion")
    A.insertion_sort();
  else
  if (algname.compare("insertion")
    A.mergesort();

  string fname_out = algname + "_" + fname_in;
  printdata(A.begin(), A.end(), fname_out);
}
```

```
-----------------------------------------------------------------
Hint: Unlike the array based implementation of mergesort which
used left and right to indicate the first and the last elements
to process, this linked list version uses left to indicate the
first element (like A.begin()) and right to be one element too
far (like A.end()).

-----------------------------------------------------------------
Hint: The node relinking done by merge (see HW5 for details) may
result in L not pointing to the first element when all is said
and done -- thus the use and return of pointers H and H->next.
-----------------------------------------------------------------
```

```
----------------------------------------------------------------
sptr_usage.cpp: smart pointer sorting using std::sort
----------------------------------------------------------------

#include <...>
using namespace std;

#include "person.h"

template <typename T>
class sptr {
  public:
    sptr(T *_ptr=NULL) { ptr = _ptr; }

    bool operator< (const sptr &rhs) const {
      return *ptr < *rhs.ptr;
    }

    operator T * () const { return ptr; }

  private:
    T *ptr;
};

template <typename T>
void data2ptr(vector<T> &A, vector < sptr<T> > &Ap) {
  Ap.resize(A.size());
  for (int i=0; i<A.size(); i++)
    Ap[i] = &A[i];
}

template <typename T>
void ptr2data(vector<T> &A, vector < sptr<T> > &Ap) {
  int i, j, nextj;

  for (i=0; i<A.size(); i++) {
    if (Ap[i] != &A[i]) {
      T tmp = A[i];
      for (j=i; Ap[j] != &A[i]; j=nextj) {
        nextj = Ap[j] - &A[0];
        A[j] = *Ap[j];
        Ap[j] = &A[j];
      }
      A[j] = tmp;
      Ap[j] = &A[j];
    }
  }
}

template <typename T>
void readdata(string &fname, vector<T> &A) { ... }

template <typename T>
void printdata(T p1, T p2, string &fname) { ... }

int main(int argc, char *argv[]) {
  if (argc != 2) {
    cerr << "usage: " << argv[0] << " file.txt\n";
    return 0;
  }

  string fname_in(argv[1]);

  vector<person_t> A;

  readdata(fname_in, A);

  vector< sptr<person_t> > Ap;

  data2ptr(A, Ap);
  std::sort(Ap.begin(), Ap.end());
  ptr2data(A, Ap);

  string fname_out = "std::sort_" + fname_in;
  printdata(A.begin(), A.end(), fname_out);
}


----------------------------------------------------------------
Hint: Sorting is carried out using an array of pointers to the
data. The sorted pointers are used to reorder the original data.

----------------------------------------------------------------
Hint: The smart pointer is a wrapper class that gives indirect
access to the data pointed to. That way, comparing two pointers
translates to comparing the underlying data.

----------------------------------------------------------------
Hint: The overloaded sptr::operator T*() allows the compiler to
make an sptr object into a pointer of type T* in case that makes
syntactic sense. Examples from ptr2data() include

  Ap[i] != &A[i]: comparison of sprt<T> and T* data
  Ap[j] - &A[0]:  pointer arithmetic applied to sptr<T> and T*
  A[j] = *Ap[j]:  dereferencing of T* in order to make result T
----------------------------------------------------------------
```