Transparency Masters

Chapter 2: step.cpp

```
// This program outputs the message
//
//    C++: one small step for the program,
//    one giant leap for the programmer
//
// to the screen
#include <iostream>
using namespace std;

int main() {
   cout << "C++: one small step for the program,\n"
        << "one giant leap for the programmer\n";
   return 0;
}
```

Figure 2.3.1: income.cpp

```
// This program repeatedly reads an income from
// the file income.in until end-of file. Income
// under 6000 greenbacks is taxed at 30 percent,
// and income greater than or equal to 6000
// greenbacks is taxed at 60 percent. After
// reading each income, the program prints the
// income and tax.
#include <fstream>
using namespace std;
const int cutoff = 6000;
const float rate1 = 0.3;
const float rate2 = 0.6;
int main() {
   ifstream infile;
   ofstream outfile;
   int income, tax;
   infile.open( "income.in" );
   outfile.open( "tax.out" );
   while ( infile >> income ) {
      if ( income < cutoff )
         tax = rate1 * income;
      else
         tax = rate2 * income;
      outfile << "Income = " << income
              << " greenbacks\n"
              << "Tax = " << tax
              << " greenbacks\n";
   }
   infile.close();
   outfile.close();
   return 0;
}
```

Figure 2.7.1: elephant.cpp

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

struct Elephant {
   string name;
   Elephant* next;
};

void      print_elephants( const Elephant* ptr );
Elephant* get_elephants();
void      free_list( const Elephant* ptr );

int main() {
   Elephant* start;
   start = get_elephants();
   print_elephants( start );
   free_list( start );
   return 0;
}

//   get_elephants dynamically allocates storage
//   for nodes. It builds the linked list and
//   stores user-supplied names in the name
//   member of the nodes. It returns a pointer
//   to the first such node.
Elephant* get_elephants() {
   Elephant *current, *first;
   int response;
   // allocate first node
   current = first = new Elephant;
   // store name of first Elephant
   cout << "\nNAME: ";
```

```
   cin >> current -> name;
   // prompt user about another Elephant
   cout << "\nAdd another? (1 == yes, 0 == no): ";
   cin >> response;
   // Add Elephants to list until user signals halt.
   while ( response == 1 ) {
      // allocate another Elephant node
      current = current -> next = new Elephant;

      // store name of next Elephant
      cout << "\nNAME: ";
      cin >> current -> name;
      // prompt user about another Elephant
      cout << "\nAdd another? (1 == yes, 0 == no): ";
      cin >> response;
   }
   // set link field in last node to 0
   current -> next = 0;
   return first;
}

//   print_elephants steps through the linked
//   list pointed to by ptr and prints the name
//   member in each node as well as the position
//   of the node in the list
void print_elephants( const Elephant* ptr ) {
   int count = 1;
   cout << "\n\n\n";
   while ( ptr !=  0 ) {
      cout << "Elephant number " << count++
           << " is " << ptr -> name << '\n';
      ptr = ptr -> next;
   }
}
```

```
//   free_list steps through the linked list pointed
//   to by ptr and frees each node in the list
void free_list( const Elephant* ptr ) {
   const Elephant* temp_ptr;
   while ( ptr !=  0 ) {
      temp_ptr = ptr -> next;
      delete ptr;
      ptr = temp_ptr;
   }
}
```

Section 3.2: A Stack Class

```
class Stack {
public:
   enum { MaxStack = 5 };
   void init() { top = -1; }
   void push( int n ) {
      if ( isFull() ) {
        errMsg( "Full stack. Can't push." );
        return;
      }
      arr[ ++top ] = n;
   }
   int pop() {
      if ( isEmpty() ) {
        errMsg( "Empty stack. Popping dummy value." );
        return dummy_val;
      }
      return arr[ top-- ];
   }
   bool isEmpty() { return top < 0; }
   bool isFull() { return top >= MaxStack - 1; }
   void dump() {
     cout << "Stack contents, top to bottom:\n";
     for ( int i = top; i >= 0; i-- )
       cout << '\t' << arr[ i ] << '\n';
   }
private:
   void errMsg( const char* msg ) const {
      cerr << "\n*** Stack operation failure: " << msg << '\n';
   }
   int top;
   int arr[ MaxStack ];
   int dummy_val;
};
```

Section 3.4: A Time Stamp Class

```
#include <iostream>
#include <ctime>
#include <string>
using namespace std;
class TimeStamp {
public:
   void set( long s = 0 ) {
      if ( s <= 0 )
        stamp = time( 0 );
      else
        stamp = s;
   }
   time_t get() const { return stamp; }
   string getAsString() const { return extract( 0, 24 ); }
   string getYear() const { return extract( 20, 4 ); }
   string getMonth() const { return extract( 4, 3 ); }
   string getDay() const { return extract( 0, 3 ); }
   string getHour() const { return extract( 11, 2 ); }
   string getMinute() const { return extract( 14, 2 ); }
   string getSecond() const { return extract( 17, 2 ); }
private:
   string extract( int offset, int count ) const {
     string timeString = ctime( &stamp );
     return timeString.substr( offset, count );
   }
   time_t stamp;
};
```

Section 3.6: A Task Class

```
#include "TimeStamp.h" //*** for TimeStamp class
#include <iostream>
#include <ctime>
#include <fstream>
#include <string>
using namespace std;
class Task {
public:
   // constructors-destructor
   Task( const string& ID ) {
      setID( ID );
      logFile = "log.dat";
      setST();
      ft = st; // no duration yet
   }
   Task( const char* ID ) {
      setID( ID );
      logFile = "log.dat";
      setST();
      ft = st; // no duration yet
   }
   ~Task() { logToFile(); }
   // set-get methods
   void setST( time_t ST = 0 ) { st.set( ST ); }
   time_t getST() const { return st.get(); }
   string getStrST() const { return st.getAsString(); }
   void setFT( time_t FT = 0 ) { ft.set( FT ); }
   time_t getFT() const { return ft.get(); }
   string getStrFT() const { return ft.getAsString(); }
   void setID( const string& ID ) { id = ID; }
   void setID( const char* ID ) { id = ID; }
   string getID() const { return id; }
   double getDU() const { return difftime( getFT(), getST() ); }
```

```
   void logToFile() {
      // set finish if duration still 0
      if ( getFT() == getST() )
        setFT();
      // log the Task's vital statistics
      ofstream outfile( logFile.c_str(), ios::app );
      outfile << "\nID: " << id << '\n';
      outfile << "  ST: " << getStrST();
      outfile << "  FT: " << getStrFT();
      outfile << "  DU: " << getDU();
      outfile << '\n';
      outfile.close(); //*** just to be safe!
   }
private:
   Task(); // default constructor explicitly hidden
   TimeStamp  st;
   TimeStamp  ft;
   string     id;
   string     logFile;
};
```

Section 4.3: Tracking Films

```
class Film {
public:
   void store_title( const string& t ) { title = t; }
   void store_title( const char* t ) { title = t; }
   void store_director( const string& d ) { director = d; }
   void store_director( const char* d ) { director = d; }
   void store_time( int t ) { time = t; }
   void store_quality( int  q ) { quality = q; }
   void output() const;
private:
   string title;
   string director;
   int time; // in minutes
   int quality; // 0 (bad) to 4 (tops)
};

void Film::output() const {
   cout << "Title: " << title << '\n';
   cout << "Director: " << director << '\n';
   cout << "Time: " << time << " mins" << '\n';
   cout << "Quality: ";
   for ( int i = 0; i < quality; i++ )
      cout << '*';
   cout << '\n';
}
```

```
class DirectorCut : public Film {
public:
   void store_rev_time( int t) { rev_time = t; }
   void store_changes( const string& s) { changes = s; }
   void store_changes( const char* s) { changes = s; }
   void output() const;
private:
   int rev_time;
   string changes;
};

void DirectorCut::output() const {
   Film::output();
   cout << "Revised time: " << rev_time << " mins\n";
   cout << "Changes: " << changes << '\n';
}

class ForeignFilm : public Film {
public:
   void store_language( const string& l) { language = l; }
   void store_language( const char* l ) { language = l; }
   void output() const;
private:
   string language;
};

void ForeignFilm::output() const {
   Film::output();
   cout << "Language: " << language << '\n';
}
```

Section 4.6: A Sequence Hierarchy

```
class Sequence {
public:
   bool addS( int, const string& );
   bool del( int );
   void output() const;
   Sequence() : last( -1 ) { }
   Sequence( const char* );
   ~Sequence();
protected:
   enum { MaxStr = 50 };
   string s[ MaxStr ];
   int last;
private:
   string filename;
   ifstream in;
   ofstream out;
};

bool Sequence::addS( int pos, const string& entry ) {
   if ( last == MaxStr - 1
        || pos < 0
        || pos > last + 1 )
      return false;
   for ( int i = last; i >= pos; i-- )
      s[ i + 1 ] = s[ i ];
   s[ pos ] = entry;
   last++;
   return true;
}
```

```
bool Sequence::del( int pos ) {
   if ( pos < 0 || pos > last )
      return false;

   for ( int i = pos; i < last; i++ )
      s[ i ] = s[ i + 1 ];
   last--;
   return true;
}

void Sequence::output() const {
   for ( int i = 0; i <= last; i++ )
      cout << i << "  " << s[ i ] << '\n';
}

Sequence::Sequence( const char* fname ) {
   last = -1;
   filename = fname;
   in.open( fname );
   if ( !in )
      return;
   while ( last < MaxStr - 1 && getline( in, s[ last + 1 ] ) )
      last++;
   in.close();
}

Sequence::~Sequence() {
   if ( filename == "" )
      return;
   out.open( filename.c_str() );
   for ( int i = 0; i <= last; i++ )
      out << s[ i ] << '\n';
   out.close();
}
```

```
class SortedSeq : public Sequence {
public:
   bool addSS( const string& );
   SortedSeq() { }
   SortedSeq( const char* );
protected:
   void sort();
private:
   using Sequence::addS;
};

void SortedSeq::sort() {
   string temp;
   int i, j;
   for ( i = 0; i <= last - 1; i++ ) {
      temp = s[ i + 1 ];
      for ( j = i; j >= 0; j-- )
         if ( temp < s[ j ] )
            s[ j + 1 ] = s[ j ];
         else
            break;
      s[ j + 1 ] = temp;
   }
}

bool SortedSeq::addSS( const string& entry ) {
   int i;
   for ( i = 0; i <= last; i++ )
      if ( entry <= s[ i ] )
         break;
   return addS( i, entry );
}
SortedSeq::SortedSeq( const char* fname ) : Sequence( fname ) {
   sort();
}
```

Section 5.2: Tracking Films Revisited

```
#include <iostream>
#include <fstream>
#include <string>
#include <cctype>
using namespace std;

class Film {
public:
   Film() {
      store_title();
      store_director();
      store_time();
      store_quality();
   }
   void store_title( const string& t ) {
      title = t;
   }
   void store_title( const char* t = "" ) {
      title = t;
   }
   void store_director( const string& d ) {
      director = d;
   }
   void store_director( const char* d = "" ) {
      director = d;
   }
   void store_time( int t = 0 ) { time = t; }
   void store_quality( int q = 0 ) { quality = q; }
   virtual void output();
   virtual void input( ifstream& );
   static bool read_input( const char* fname, Film*[ ], int );
private:
   string title;
```

```
   string director;
   int time;   // in minutes
   int quality; // 0 (bad) to 4 (tops)
};

// Reads title, director, time, and quality.
void Film::input( ifstream& fin ) {
   string inbuff;
   getline( fin, inbuff );
   store_title( inbuff );
   getline( fin, inbuff );
   store_director( inbuff );
   getline( fin, inbuff );
   store_time( atoi( inbuff.c_str() ) );
   getline( fin, inbuff);
   store_quality( atoi( inbuff.c_str() ) );
}

// Writes title, director, time, and quality.
void Film::output() {
   cout << "Title: " << title << endl;
   cout << "Director: " << director << endl;
   cout << "Time: " << time << " mins" << endl;
   cout << "Quality: ";
   for ( int i = 0; i < quality; i++ )
      cout << '*';
   cout << endl;
}

class DirectorCut : public Film {
public:
   DirectorCut() {
      store_rev_time();
      store_changes();
   }
```

```
   void store_rev_time( int t = 0 ) { rev_time = t; }
   void store_changes( const string& c ) { changes = c; }
   void store_changes( const char* c = "" ) { changes = c; }
   virtual void output();
   virtual void input( ifstream& );
private:
   int rev_time;
   string changes;
};

// Reads revised time and changes.
void DirectorCut::input( ifstream& fin ) {
   Film::input( fin );
   string inbuff;
   getline( fin, inbuff );
   store_rev_time( atoi( inbuff.c_str() ) );
   getline( fin, inbuff );
   store_changes( inbuff );
}

// Writes revised time and changes.
void DirectorCut::output() {
   Film::output();
   cout << "Revised time: " << rev_time << endl;
   cout << "Changes: " << changes << endl;
}

class ForeignFilm : public Film {
public:
   ForeignFilm() { store_language(); }
   void store_language( const string& l ) { language = l; }
   void store_language( const char* l = "" ) { language = l; }
   virtual void output();
   virtual void input( ifstream& );
private:
```

```
   string language;
};

// Reads language.
void ForeignFilm::input( ifstream& fin ) {
   Film::input( fin );
   string inbuff;
   getline( fin, inbuff );
   store_language( inbuff );
}

// Writes language.
void ForeignFilm::output() {
   Film::output();
   cout << "Language: " << language << endl;
}

// class method: Film::read_input
// Reads data from an input file, dynamically creating the
// appropriate Film object for each record group. For instance,
// a ForeignFilm object is dynamically created if the data
// represent a foreign film rather than a regular film or a
// director's cut. Pointers to dynamically created objects are
// stored in the array films of size n. Returns true to signal
// success and false to signal failure.
bool Film::read_input( const char* file,
                       Film* films[ ], int n ) {
   string inbuff;
   ifstream fin( file );
   if ( !fin )  // opened successfully?
     return false; // if not, return false

   // Read until end-of-file. Records fall into
   // groups. 1st record in each group is a string
   // that represents a Film type:
```

```
   //   "Film", "ForeignFilm", "DirectorCut", etc.
   // After reading type record, dynamically create
   // an object of the type (e.g., a ForeignFilm object),
   // place it in the array films, and invoke its
   // input method.
   int next = 0;
   while ( getline( fin, inbuff ) && next < n ) {
      if ( inbuff == "Film" )
        films[ next ] = new Film();        // regular film
      else if ( inbuff == "ForeignFilm" )
        films[ next ] = new ForeignFilm(); // foreign film
      else if ( inbuff == "DirectorCut" )
        films[ next ] = new DirectorCut(); // director's cut
      else //**** error condition: unrecognized film type
        continue;
      films[ next++ ]->input( fin ); // polymorphic method
   }
   fin.close();
   return true;
}
```

Section 6.2: A Complex Number Class

```
class Complex {
public:
   Complex();                   // default
   Complex( double );           // real given
   Complex( double, double ); // both given

   void write() const;
   // operator methods
   Complex operator+( const Complex& ) const;
   Complex operator-( const Complex& ) const;
   Complex operator*( const Complex& ) const;
   Complex operator/( const Complex& ) const;
private:
   double real;
   double imag;
};

// default constructor
Complex::Complex() {
   real = imag = 0.0;
}

// constructor -- real given but not imag
Complex::Complex( double re ) {
   real = re;
   imag = 0.0;
}

// constructor -- real and imag given
Complex::Complex( double re, double im ) {
   real = re;
   imag = im;
}
```

```
void Complex::write() const {
   cout << real << " + " << imag << 'i';
}

// Complex + as binary operator
Complex Complex::operator+( const Complex& u ) const {
   Complex v( real + u.real,
              imag + u.imag );
   return v;
}

// Complex - as binary operator
Complex Complex::operator-( const Complex& u ) const {
   Complex v( real - u.real,
              imag - u.imag );
   return v;
}

// Complex * as binary operator
Complex Complex::operator*( const Complex& u ) const {
   Complex v( real * u.real - imag * u.imag,
              imag * u.real + real * u.imag );
   return v;
}

// Complex / as binary operator
Complex Complex::operator/( const Complex& u ) const {
   double abs_sq = u.real * u.real + u.imag * u.imag;
   Complex v( ( real * u.real + imag * u.imag ) / abs_sq,
              ( imag * u.real - real * u.imag ) / abs_sq );
   return v;
}
```

Section 6.8: An Associative Array

```
class Entry {
public:
   Entry() { flag = false; }
   void add( const string&, const string& );
   bool match( const string& ) const;
   void operator=( const string& );
   void operator=( const char* );
   friend ostream& operator<<( ostream&, const Entry& );
   bool valid() const { return flag; }
private:
   string word;
   string def;
   bool flag;
};
void Entry::operator=( const string& str ) {
   def = str;
   flag = true;
}
void Entry::operator=( const char* str ) {
   def = str;
   flag = true;
}

ostream& operator<<( ostream& out, const Entry& e ) {
   out << e.word << " defined as: "
       << e.def;
   return out;
}

void Entry::add( const string& w, const string& d ) {
   word = w;
   def = d;
}
```

```
bool Entry::match( const string& key ) const {
   return key == word;
}

class Dict {
public:
   enum { MaxEntries = 100 };
   friend ostream& operator<<( ostream&, const Dict& );
   Entry& operator[ ]( const string& );
   Entry& operator[ ]( const char* );
private:
   Entry entries[ MaxEntries + 1 ];
};

ostream& operator<<( ostream& out, const Dict& d ) {
   for ( int i = 0; i < MaxEntries; i++ )
      if ( d.entries[ i ].valid() )
         out << d.entries[ i ] << '\n';
   return out;
}

Entry& Dict::operator[ ]( const string& k ) {
   for ( int i = 0; i < MaxEntries && entries[ i ].valid();
         i++ )
      if ( entries[ i ].match( k ) )
         return entries[ i ];
   string not_found = "*** not in dictionary";
   entries[ i ].add( k, not_found );
   return entries[ i ];
}
Entry& Dict::operator[ ]( const char* k ) {
   string s = k;
   return operator[ ]( s );
}
```

Section 7.2: A Template Stack Class

```
#include <iostream>
#include <string>
//#define NDEBUG //**** enable/disable assertions
#include <cassert>
using namespace std;
template< class T >
class Stack {
public:
   enum { DefaultStack = 50, EmptyStack = -1 };
   Stack();
   Stack( int );
   ~Stack();
   void push( const T& );
   T pop();
   T topNoPop() const;
   bool empty() const;
   bool full() const;
private:
   T* elements;
   int top;
   int size;
   void allocate() {
     elements = new T[ size ];
     top = EmptyStack;
   }
   void msg( const char* m ) const {
     cout << "*** " << m << " ***" << endl;
   }
   friend ostream& operator<<( ostream&, const Stack< T >& );
};
```

```
template< class T >
Stack< T >::Stack() {
   size = DefaultStack;
   allocate();
}

template< class T >
Stack< T >::Stack( int s ) {
   if ( s < 0 )       // negative size?
     s *= -1;
   else if ( 0 == s ) // zero size?
     s = DefaultStack;
   size = s;
   allocate();
}

template< class T >
Stack< T >::~Stack() {
   delete[ ] elements;
}

template< class T >
void Stack< T >::push( const T& e ) {
   assert( !full() );
   if ( !full() )
     elements[ ++top ] = e;
   else
     msg( "Stack full!" );
}

template< class T >
T Stack< T >::pop() {
   assert( !empty() );
   if ( !empty() )
     return elements[ top-- ];
```

```
   else {
     msg( "Stack empty!" );
     T dummy_value;
     return dummy_value; // return arbitrary value
   }
}

template< class T >
T Stack< T >::topNoPop() const {
   assert( top > EmptyStack );
   if ( !empty() )
     return elements[ top ];
   else {
     msg( "Stack empty!" );
     T dummy_value;
     return dummy_value;
   }
}

template< class T >
bool Stack< T >::empty() const {
   return top <= EmptyStack;
}
template< class T >
bool Stack< T >::full() const {
   return top + 1 >= size;
}
template< class T >
ostream& operator<<( ostream& os, const Stack< T >& s ) {
   s.msg( "Stack contents:" );
   int t = s.top;
   while ( t > s.EmptyStack )
     cout << s.elements[ t-- ] << endl;
   return os;
}
```

Section 7.4: Stock Performance Reports

```
#include <iostream>
#include <fstream>
#include <deque>
#include <algorithm>
#include <string>
using namespace std;

//*** file names and miscellaneous globals
const string inFile = "stockData.dat";
const string Unknown = "????";

//*** objects generated from input records
class Stock {
public:
   Stock() {
      symbol = Unknown;
      open = close = gainLoss = volume = 0;
   }
   Stock( const string& s,     // symbol
          double o,            // opening price
          double c,            // closing price
          unsigned long v ) {  // volume traded
      symbol = s;
      open = o;
      close = c;
      volume = v;
      gainLoss = ( close - open ) / open;
   }

   const string& getSymbol() const {
      return symbol;
   }
```

```
   double getOpen() const {
      return open;
   }
   double getClose() const {
      return close;
   }
   unsigned long getVolume() const {
      return volume;
   }
   double getGainLoss() const {
      return gainLoss;
   }
private:
   string        symbol;
   double        open;     // opening price
   double        close;    // closing price
   double        gainLoss; // gain or loss fraction
   unsigned long volume;   // shares traded
};

//*** Sort comparison: gains in descending order
struct winCmp {
   bool operator()( const Stock& s1, const Stock& s2 ) const  {
      return s1.getGainLoss() > s2.getGainLoss();
   }
};

//*** Sort comparison: volume in descending order
struct volCmp {
   bool operator()( const Stock& s1, const Stock& s2 ) const {
      return s1.getVolume() > s2.getVolume();
   }
};
```

```
//*** invoked by function objects to do output
void output( bool volFlag,
             const string& name,
             const char* openLabel, double open,
             const char* closeLabel, double close,
             const char* gainLabel, double gain,
             const char* volLabel, unsigned long vol ) {
   cout << "*** " << name << endl;
   if ( volFlag ) // if true, volume comes first
     cout << '\t' << volLabel << vol << endl;
   cout << '\t' << gainLabel  << gain  << endl
        << '\t' << openLabel  << open  << endl
        << '\t' << closeLabel << close << endl;
   if ( !volFlag ) // if false, volume comes last
     cout << '\t' << volLabel << vol << endl;
}

//*** Write Stocks sorted by gain-loss to standard output.
struct winPr {
   void operator()( const Stock& s ) const {
      output( false,
              s.getSymbol(),
              "Opening Price: ", s.getOpen(),
              "Closing Price: ", s.getClose(),
              "% Changed:     ", s.getGainLoss() * 100,
              "Volume:        ", s.getVolume() );
   }
};

//*** Write Stocks sorted by volume to standard output.
struct volPr {
   void operator()( const Stock& s ) const {
      output( true,
              s.getSymbol(),
              "Opening Price: ", s.getOpen(),
```

```
              "Closing Price: ", s.getClose(),
              "% Changed:     ", s.getGainLoss() * 100,
              "Volume:        ", s.getVolume() );
   }
};

void herald( const char* );
void input( deque< Stock >& );
int main() {
   deque< Stock > stocks;
   //*** Input stocks and separate into vectors for
   //    winners, losers, and break-evens.
   input( stocks );
   //*** Sort winners in descending order and output.
   herald( "Gainers in descending order: " );
   sort( stocks.begin(), stocks.end(), winCmp() );
   for_each( stocks.begin(), stocks.end(), winPr() );
   //*** Output losers in ascending order.
   herald( "Losers in ascending order: " );
   for_each( stocks.rbegin(), stocks.rend(), winPr() );
   //*** Sort volume in descending order and output
   herald( "Volume in descending order: " );
   sort( stocks.begin(), stocks.end(), volCmp() );
   for_each( stocks.begin(), stocks.end(), volPr() );
   return 0;
}
```

```
void input( deque< Stock >& d ) {
   string s;
   double o, c, v;
   ifstream input( inFile.c_str() );
   //*** Read data until end-of-file,
   // creating a Stock object per input record
   while ( input >> s >> o >> c >> v )
     d.insert( d.end(), Stock( s, o, c, v ) );
   input.close();
}

void herald( const char* s ) {
   cout << endl << "******* " << s << endl;
}
```

Section 8.6: A Random Access File Class

```
#include <iostream>
#include <cstdio>
#include <fstream>
#include <cstring>
#include <cstdlib>
#include <cctype>
using namespace std;

const int header_size = 256;
const char Taken = 'T';
const char Free = 'F';
const char Deleted = 'D';

class frandom : public fstream {
public:
   frandom();
   frandom( const char* ); // open existing file
   frandom( const char*, int, int, int ); // open new file
   ~frandom();
   void open( const char* ); // open existing file
   void open( const char*, int, int, int ); // open new file
   void close();
   long get_slots() const { return slots; }
   long get_record_size() const { return record_size; }
   long get_key_size() const { return key_size; }
   long get_total_bytes() const { return total_bytes; }
   long get_no_records() const { return no_records; }
   bool add_record( const char* );
   bool find_record( char* );
   bool remove_record( const char* );
private:
   long slots;
   long record_size; // includes 1-byte flag
```

```
   long key_size;
   long total_bytes;
   long no_records; // no of records stored
   long loc_address; // computed by locate
   char* buffer; // holds one record
   char* stored_key; // holds one key
   long get_address( const char* ) const;
   bool locate( const char* );
};

frandom::~frandom() {
   if ( is_open() ) {
      delete [ ] stored_key;
      delete [ ] buffer;
      char buff[ header_size ];
      for ( int i = 0; i < header_size; i++ )
         buff[ i ] = ' ';
      sprintf( buff, "%ld %ld %ld %ld",
               slots, record_size,
               key_size, no_records );
      seekp( 0, ios_base::beg );
      write( buff, header_size );
   }
}

frandom::frandom() : fstream() {
   buffer = stored_key = 0;
   slots = record_size = key_size = 0;
   total_bytes = no_records = 0;
}

frandom::frandom( const char* filename ) : fstream() {
   buffer = stored_key = 0;
   open( filename );
}
```

```
frandom::frandom( const char* filename,
                  int sl,
                  int actual_record_size,
                  int ks ) : fstream() {
   buffer = stored_key = 0;
   open( filename, sl, actual_record_size, ks );
}

// open an existing file
void frandom::open( const char* filename ) {
   fstream::open( filename,
                  ios_base::in |
                  ios_base::out |
                  ios_base::binary );

   if ( is_open() ) {
      char buff[ header_size ];
      read( buff, header_size );
      sscanf( buff, "%ld%ld%ld%ld",
              &slots, &record_size, &key_size,
              &no_records );
      total_bytes = slots * record_size + header_size;
      // get_address needs \0
      stored_key = new char [ key_size + 1 ];
      buffer = new char [ record_size ];
   }
}
```

```
// open a new file
void frandom::open( const char* filename,
                    int sl,
                    int actual_record_size,
                    int ks ) {
   fstream::open( filename,
                  ios_base::in |
                  ios_base::out |
                  ios_base::binary );
   // if open succeeds, file already exists
   if ( is_open() ) {
      setstate( ios_base::failbit );
      fstream::close();
      return;
   }

   // file does not exist; create it
   fstream::open( filename, ios_base::out | ios_base::binary );
   if ( is_open() )
      fstream::close();

   // file created; now open it for input and output
   fstream::open( filename,
                  ios_base::in |
                  ios_base::out |
                  ios_base::binary );

   if ( is_open() ) {
      clear(); // clear failbit flag set by open failure
      char buff[ header_size ];
      slots = sl;
      record_size = actual_record_size + 1;
      key_size = ks;
      total_bytes = slots * record_size + header_size;
      no_records = 0;
```

```
      // get_address needs \0
      stored_key = new char [ key_size + 1 ];
      for ( int i = 0; i < header_size; i++ )
         buff[ i ] = ' ';
      sprintf( buff, "%ld %ld %ld %ld",
               slots, record_size,
               key_size, no_records );
      write( buff, header_size );
      buffer = new char [ record_size ];
      for ( i = 1; i < record_size; i++ )
         buffer[ i ] = ' ';
      buffer[ 0 ] = Free;
      for ( i = 0; i < slots; i++ )
         write( buffer, record_size );
   }
}

// hash function
long frandom::get_address( const char* key ) const {
   memcpy( stored_key, key, key_size );
   stored_key[ key_size ] = '\0';
   return ( atol( stored_key ) % slots )
            * record_size + header_size;
}

// locate searches for a record with the specified key.
// If successful, locate returns true.
// If unsuccessful, locate returns false.
// locate sets data member loc_address to the address
// of the record if the record is found. This
// address can be then used by find_record
// or remove_record.
//
// If the record is not found, locate sets loc_address
// to the first D or F slot encountered in its search
```

```
// for the record. This address can be used by add_record.
// If there is no D or F slot (full file), locate sets
// loc_address to the hash address of key.
bool frandom::locate( const char* key ) {
   // address = current offset in file
   // start_address = hash offset in file
   // unocc_address = first D slot in file
   //               = start_address, if no D slot
   long address, start_address, unocc_address;

   // delete_flag = false, if no D slot is found
   //             = true, if D slot is found
   int delete_flag = false;

   address = get_address( key );
   unocc_address = start_address = address;
   do {
      seekg( address, ios_base::beg );
      switch( get() ) {
      case Deleted:
         if ( !delete_flag ) {
            unocc_address = address;
            delete_flag = true;
         }
         break;
      case Free:
         loc_address = delete_flag ? unocc_address : address;
    return false;
      case Taken:
         seekg( address + 1, ios_base::beg );
         read( stored_key, key_size );
         if ( memcmp( key, stored_key, key_size ) == 0 ) {
            loc_address = address;
            return true;
         }
```

```
         break;
      }
      address += record_size;
      if ( address >= total_bytes )
         address = header_size;
   } while ( address != start_address );

   loc_address = unocc_address;
   return false;
}

bool frandom::add_record( const char* record ) {
   if ( no_records >= slots || locate( record ) )
      return false;

   seekp( loc_address, ios_base::beg );
   write( &Taken, 1 );
   write( record, record_size - 1 );
   no_records++;
   return true;
}

bool frandom::find_record( char* record ) {
   if ( locate( record ) ) {
      seekg( loc_address + 1, ios_base::beg );
      read( record, record_size - 1 );
      return true;
   }
   else
      return false;
}

bool frandom::remove_record( const char* key ) {
   if ( locate( key ) ) {
      --no_records;
```

```
      seekp( loc_address, ios_base::beg );
      write( &Deleted, 1 );
      return true;
   }
   else
      return false;
}

void frandom::close() {
   if ( is_open() ) {
      delete [ ] stored_key;
      delete [ ] buffer;
      char buff[ header_size ];
      for ( int i = 0; i < header_size; i++ )
         buff[ i ] = ' ';
      sprintf( buff, "%ld %ld %ld %ld",
               slots, record_size,
               key_size, no_records );
      seekp( 0, ios_base::beg );
      write( buff, header_size );
      fstream::close();
   }
}

int main() {
   char b[ 10 ], c;

   frandom finout;

   cout << "New file (Y/N)? ";
   cin >> c;
   if ( toupper( c ) == 'Y' ) {
      finout.open( "data.dat", 15, 5, 3 );
      if ( !finout ) {
         cerr << "Couldn't open file\n";
```

```
         return EXIT_FAILURE;
      }
   }
   else {
      finout.open( "data.dat" );
      if ( !finout ) {
         cerr << "Couldn't open file\n";
         return EXIT_FAILURE;
      }
   }

   do {
      cout << "\n\n[A]dd\n[F]ind\n[R]emove\n[Q]uit? ";
      cin >> c;
      switch ( toupper( c ) ) {
      case 'A':
         cout << "Which record to add? ";
         cin >> b;
         if ( finout.add_record( b ) )
            cout << "Record added\n";
         else
            cout << "Record not added\n";
         break;
      case 'F':
         cout << "Key? ";
         cin >> b;
         if ( finout.find_record( b ) ) {
            b[ 5 ] = '\0';
            cout << "Record found: " << b << '\n';
         }
         else
            cout << "Record not found\n";
         break;
      case 'R':
         cout << "Key? ";
```

```
        cin >> b;
        if ( finout.remove_record( b ) )
           cout << "Record removed\n";
        else
           cout << "Record not removed\n";
        break;
     case 'Q':
        break;
     default:
        cout << "Illegal choice\n";
        break;
     }
  } while ( toupper( c ) != 'Q' );

  return 0;
}
```

Section 8.8: A High-Level Copy Function

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

// copy from basic_istream to basic_ostream
template< class charT, class traits>
void copy( basic_istream<charT, traits>& in,
           basic_ostream<charT, traits>& out ) {
   charT c;
   ios_base::iostate fl;
   // don't skip white space
   // save old flags in fl to restore at end
   fl = in.flags();
   in.unsetf( ios_base::skipws );
   while ( in >> c )
      out << c;
   // restore flags
   in.flags( fl );
}
```