Encapsulation

Steven Zeil

July 17, 2013

Contents

1 Encapsulation

The ADT as Contract

When an ADT specification/implementation is provided to users (application programmers):

- Users are expected to alter/examine values of this type only via the members specified.
- The creator of the ADT promises to leave the member specifications unchanged.

. . . . . .

Enforcing the Contract

- If we leave it like this (10), then the contract is just a "gentlemen's agreement".
  - Programmers in older programming languages do just that.
  - But programming language designers eventually noticed.
- *Encapsulation* is the use of programming language rules to enforce information hiding design decisions.

. . . . . .

1.1 Encapsulation in C++

Encapsulation in C++

We can mark portions of a `struct` as

- *public*, meaning that names declared there can be used by any code.
- *private*, meaning that names declared there can only be used by code that is part of the struct.

. . . . . .

Time for Encapsulation

```
struct Time {
public:
  // Create time objects
  Time(); // midnight
  Time (int h, int m, int s);

  // Access to attributes
```

```
  int getHours();
  int getMinutes();
  int getSeconds();

  // Calculations with time
  void add (Time delta);
  Time difference (Time fromTime);

  /**
   * Read a time (in the format HH:MM:SS) after skipping any
   * prior whitepace.
   */
  void read (std::istream& in);


  /**
   * Print a time in the format HH:MM:SS (two digits each)
   */
  void print (std::ostream& out);

  /**
   *  Compare two times. Return true iff time1 is earlier than or equal to
   *  time2
   */
  bool noLaterThan(const Time& time2);


  /**
   *  Compare two times. Return true iff time1 is equal to
   *  time2
   *
   */
  bool equalTo(const Time& time2);

private:
  // From here on is hidden
  int secondsSinceMidnight;
};
```

```
int Time::getHours()
```


```
{
  return secondsSinceMidnight / 3600; // OK
}

int getHours(Time t)
{
  return t.secondsSinceMidnight / 3600; // Error
}
```

. . . . . .

2 Classes

From structs to classes

- Most ADTs in C++ are written as `classes`, not `structs`.
- Almost identical, according to language rules

  Until you explicitly say "public:" or "private:",
  - Structs start out as "public"
  - Classes start out as "private"
- Very different in style/idiom
  - Classes are used for "real" ADTs that the application coders will work with.
  - Structs are used for "helper" types that the classes uses to get their work done
    - often nested inside classes

    or for such trivial structured data that no information hiding is necessary

. . . . . .

Showing a Little Class

Our time class:

```
class Time {
public:
  // Create time objects
  Time(); // midnight
  Time (int h, int m, int s);

  // Access to attributes
```

```
  int getHours();
  int getMinutes();
  int getSeconds();

  // Calculations with time
  void add (Time delta);
  Time difference (Time fromTime);

  /**
   * Read a time (in the format HH:MM:SS) after skipping any
   * prior whitepace.
   */
  void read (std::istream& in);


  /**
   * Print a time in the format HH:MM:SS (two digits each)
   */
  void print (std::ostream& out);

  /**
   *  Compare two times. Return true iff time1 is earlier than or equal to
   *  time2
   */
  bool noLaterThan(const Time& time2);


  /**
   *  Compare two times. Return true iff time1 is equal to
   *  time2
   *
   */
  bool equalTo(const Time& time2);

private:
  // From here on is hidden
  int secondsSinceMidnight;
};
```

And our alternate version of the same class:

```
class Time {
  // The declaration below is hidden
   int secondsSinceMidnight;
public:
  // Create time objects
  Time(); // midnight
  Time (int h, int m, int s);

  // Access to attributes
  int getHours();
  int getMinutes();
  int getSeconds();

  // Calculations with time
  void add (Time delta);
  Time difference (Time fromTime);

  /**
   * Read a time (in the format HH:MM:SS) after skipping any
   * prior whitepace.
   */
  void read (std::istream& in);


  /**
   * Print a time in the format HH:MM:SS (two digits each)
   */
  void print (std::ostream& out);

  /**
   *  Compare two times. Return true iff time1 is earlier than or equal to
   *  time2
   */
  bool noLaterThan(const Time& time2);


  /**
   *  Compare two times. Return true iff time1 is equal to
   *  time2
   *
```

```
   */
  bool equalTo(const Time& time2);
};
```

. . . . . .

3 Hiding Attributes

Hide Your Attributes

An almost universal rule of thumb for ADTs in C++:

> All data members should be private.

- Attributes are data values that we regard as "contained" in our abstraction
  - They are part of the mental model, must be part of the interface
- If a data member represents an attribute, can we hide it?
  - Yes, but we provide access via get/set functions.

. . . . . .

Example: Time Attributes

```
class Time {
public:
  // Create time objects
  Time(); // midnight
  Time (int h, int m, int s);

  // Access to attributes
  int getHours();
  void setHours (int h);
  int getMinutes();
  void setMinutes (int m);
  int getSeconds();
  void setSeconds (int s);
  // Calculations with time
  void add (Time delta);
  Time difference (Time fromTime);
```

```
  /**
   * Read a time (in the format HH:MM:SS) after skipping any
   * prior whitepace.
   */
  void read (std::istream& in);


  /**
   * Print a time in the format HH:MM:SS (two digits each)
   */
  void print (std::ostream& out);

  /**
   *  Compare two times. Return true iff time1 is earlier than or equal to
   *  time2
   */
  bool noLaterThan(const Time& time2);


  /**
   *  Compare two times. Return true iff time1 is equal to
   *  time2
   *
   */
  bool equalTo(const Time& time2);
private:
  // From here on is hidden
  int hours;
  int minutes;
  int seconds;

};
```

```
int Time::getHours()
{
  return hours;
}

void Time::setHours(int h)
{
```

```
  hours = h;
}
```

. . . . . .

4 Inline Functions

Are Short Functions Ineffcient?

Are ADT implementations terribly inefficient?

They tend to feature a lot of one-liners and similar short functions.

```
int Time::getHours()
{
  return hours;
}
```

. . . . . .

4.1 How Functions Work

How Functions Work

```
int foo(int a, int b)
{
  return a+b-1;
}
```

would compile into a block of code equivalent to

```
  stack[1] = stack[3] + stack[2] - 1;
  jump to address in stack[0]
```

. . . . . .

The Runtime Stack

- the "stack" is the *runtime stack* (a.k.a. the *activation stack*) used to track function calls at the system level,
- stack[0] is the top value on the stack,
- stack[1] the value just under that one, and so on.

. . . . . .

An Example of Function Activation

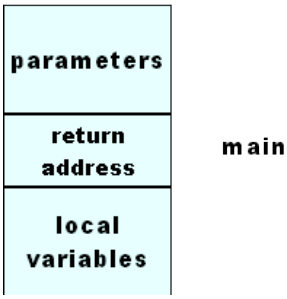
Suppose that we were executing this code, and had just come to the call to resolveAuction within `main`.

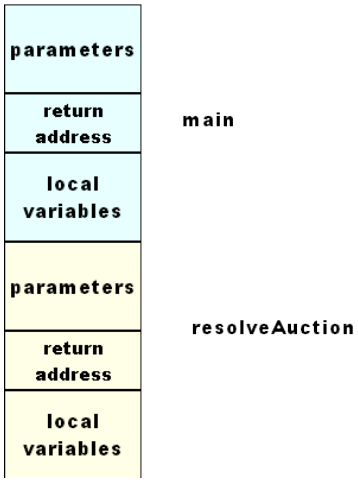
```
#include "time.h"

void resolveAuction (Item item)
{
  ⋮
  int h = item.auctionEndsAt.getHours();
  ⋮
}

int main (int argc, char** argv)
{
  ⋮
  resolveAuction (item);
  ⋮
}
```

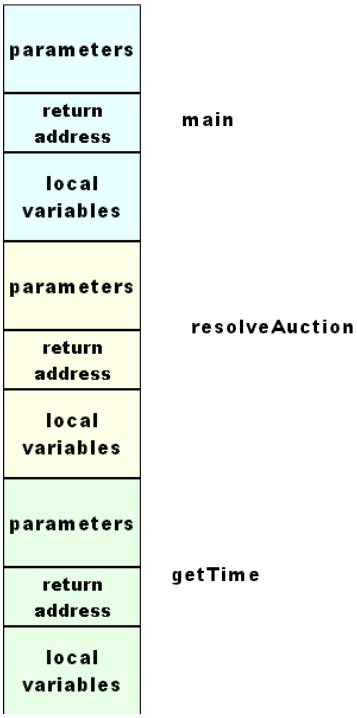
The runtime stack (a.ka., the activation stack) would, at this point in time, contain a single *activation record* for the `main` function, as that is the only function currently executing:

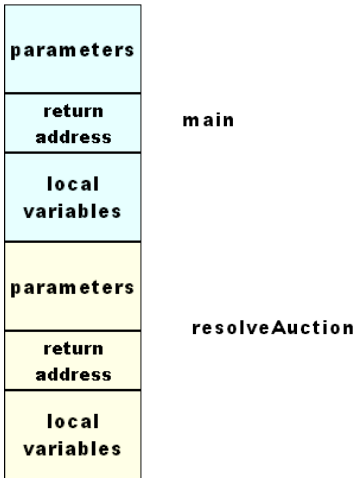




When `main` calls `resolveAuction`, a new record is added to the stack with space for all the data required for this new function call.

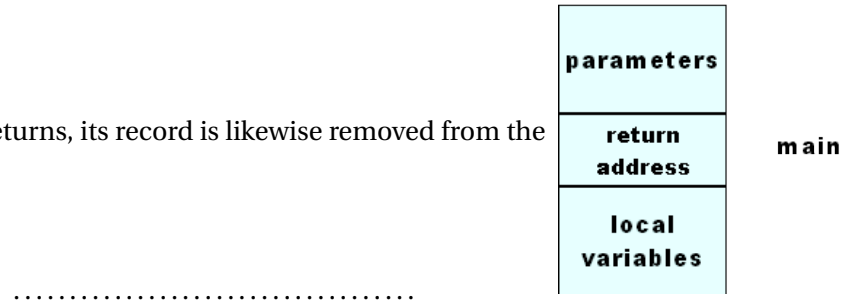
When `resolveAuction` calls `getHours`, another new record is added to the stack with space for all the data required for *that* new function call.





But once `getHours` returns (to `resolveAuction`), that activation record is removed from the stack. `resolveAuction` is once again the active function.

And when `resolveAuction` returns, its record is likewise removed from the stack.



4.2 How Function Calls Work

How Function Calls Work

A function call like

```
x = foo(y,z+1);
```

would be compiled into a code sequence along the lines of

```
    push y onto the runtime stack;
    evaluate z+1;
    push the result onto the runtime stack
    push (space for the return value) onto the runtime stack
    save all CPU registers
    push address RET onto the runtime stack
    jump to start of foo's body
RET: x = stack[1]
    pop runtime stack 4 times
    restore all CPU registers
```

. As you can see, there's a fair amount of overhead involved in passing parameters and return address information to a function when making a call.

- The amount of time spent on this overhead is really all that large.
- If the function body contains several statements of any kind of loop, then the overhead is probably a negligable fraction of the total time spent on the call.

Overhead

```
int Time::getHours()
{
  return hours;
}

void Time::setHours(int h)
```

```
{
  hours = h;
}
```

- For functions like these, the overhead may be greater than the time needed for the function body itself
- If called inside a loop that repeats many times, delay can be significant

.....

4.3 Inline Functions

Inline Functions

```
class Time {
public:
  // Create time objects
  Time(); // midnight
  Time (int h, int m, int s);

  // Access to attributes
   int getHours() {return hours;}
   void setHours (int h) {hours = h;}
  int getMinutes();
  void setMinutes (int m);
    ⋮
private:
  int hours;
  int minutes;
  int seconds;

};

inline
int Time::getMinutes()
{
   return minutes;
}
```

```
inline
void Time::setMinutes(int m)
{
  minutes = m;
}
```

- C++ offers the option of declaring functions as *inline*.
  - can be written one of two ways.
    * inside the class declaration.
    * or place the reserved word *inline* in front of the function definition written outside the class declaration.

. . . . . .

How Inline Functions Work

When we make a call to an inline function, the compiler simply replaces the call by a compiled copy of the function body (with some appropriate renaming of variables to avoid conflicts).

. . . . . .

Example of an Inline Call

If we have

```
inline int foo(int a, int b)
{
  return a+b-1;
}
```

and we later make a call

```
  x = foo(y,z+1);
```

This would be compiled into a code sequence along the lines of

```
   evaluate z+1, storing result in tempB
   evaluate y + tempB - 1, storing result in x
```

Most of the overhead of making a function call has been eliminated.

. . . . . .

Use Inline For Functions That Are...

- short

- called many times

If abused, inline calls tend to make the size of the executable program grow.

. . . . . .

Inlining is Optional

Inlining is only a recommendation from the programmer to the compiler.

- The compiler may ignore an inline declaration if it prefers.
  - inlining of functions with recursive calls is impossible
  - Many compilers will refuse to inline any function whose body contains a loop.

. . . . . .

5 Example: the Auction Program as Encapsulated ADTs

The Auction Program

- ```
  #include <iostream>
  #include <string>

  using namespace std;

  #include "items.h"
  #include "itemcollection.h"
  #include "bidders.h"
  #include "biddercollection.h"
  #include "bids.h"
  #include "bidcollection.h"
  #include "time.h"

  const int MaxBidders = 100;
  const int MaxBids = 5000;
  const int MaxItems = 100;


  /**
   * Determine the winner of the auction for item #i.
   * Announce the winner and remove money from winner's account.
   */
  ```
  ```

```
void resolveAuction (const Item& item,
                     BidderCollection& bidders,
                     BidCollection& bids);


int main (int argc, char** argv)
{
  if (argc != 4)
    {
      cerr << "Usage: " << argv[0] << " itemsFile biddersFile bidsFile" << endl;
      return -1;
    }

  ItemCollection items (MaxItems);
  items.readItems (argv[1]);

  BidderCollection bidders (MaxBidders);
  bidders.readBidders (argv[2]);

  BidCollection bids (MaxBids);
  bids.readBids (argv[3]);

  for (int i = 0; i < items.getSize(); ++i)
    {
      resolveAuction(items.get(i), bidders, bids);
    }
  return 0;
}


/**
 * Determine the winner of the auction for an item.
 * Announce the winner and remove money from winner's account.
 */
void resolveAuction (const Item& item,
                     BidderCollection& bidders,
                     BidCollection& bids)
{
  double highestBidSoFar = 0.0;
  string winningBidderSoFar;
```

```
bool reservePriceMet = false;
for (int bidNum = 0; bidNum < bids.getSize(); ++bidNum)
  {
    Bid bid = bids.get(bidNum);
    if (bid.getTimePlacedAt().noLaterThan(item.getAuctionEndTime()))
      {
        if (bid.getItem() == item.getName()
            && bid.getAmount() > highestBidSoFar
            )
          {
            int bidderNum = bidders.findBidder(bid.getBidder());
            Bidder bidder = bidders.get(bidderNum);
            // Can this bidder afford it?
            if (bid.getAmount() <= bidder.getBalance())
              {
                highestBidSoFar = bid.getAmount();
                winningBidderSoFar = bid.getBidder();
              }
          }
          if (bid.getAmount() > item.getReservedPrice())
              reservePriceMet = true;
      }
  }

  // If highestBidSoFar is non-zero, we have a winner
  if (reservePriceMet && highestBidSoFar > 0.0)
    {
      int bidderNum = bidders.findBidder(winningBidderSoFar);
      cout << item.getName()
           << " won by " << winningBidderSoFar
           << " for " << highestBidSoFar << endl;
      Bidder& bidder = bidders.get(bidderNum);
      bidder.setBalance (bidder.getBalance() - highestBidSoFar);
    }
  else
    {
      cout << item.getName()
           << " reserve not met"
           << endl;
    }
```


```
}
```

- The Bid ADT:

```
#ifndef BIDS_H
#define BIDS_H

#include <string>
#include "time.h"


//
//  Bids Received During Auction
//

class Bid {

  std::string bidderName;
  double amount;
  std::string itemName;
  Time bidPlacedAt;

public:

  Bid (std::string bidder, double amt,
       std::string item, Time placedAt);

  Bid();

  // Access to attribute
  std::string getBidder() const {return bidderName;}
  double getAmount() const {return amount;}
  std::string getItem() const {return itemName;}
  Time  getTimePlacedAt() const {return bidPlacedAt;}

};
```

```
#endif
```

and

```
#include <string>
#include <fstream>

using namespace std;

//
//  Bids Received During Auction
//

#include "bids.h"


Bid::Bid (std::string bidder, double amt,
     std::string item, Time placedAt)
  : bidderName(bidder), amount(amt),
    itemName(item), bidPlacedAt(placedAt)
{
}

Bid::Bid ()
  : amount(0.0)
{
}
```

- The Bidder ADT:

```
#ifndef BIDDERS_H
#define BIDDERS_H

#include <string>

//
//  Bidders Registered for auction
//
```

```
class Bidder {
  // describes someone registered to participate in an auction
  std::string name;
  double balance;

public:

  Bidder();
  Bidder (std::string theName, double theBalance);

  // Access to attributes
  std::string getName() const {return name;}

  double getBalance() const {return balance;}
  void setBalance (double newBal) {balance = newBal;}
};


#endif
```

and

```
#include <string>
#include <fstream>
#include <iostream>
//
//  Bidders Registered for auction
//

#include "bidders.h"

using namespace std;



Bidder::Bidder()
{
  name = "";
```

```
  balance = 0.0;
}

Bidder::Bidder (std::string theName, double theBalance)
{
  name = theName;
  balance = theBalance;
}
```

- The Item ADT:

```
#ifndef ITEMS_H
#define ITEMS_H

#include <iostream>
#include <string>

#include "time.h"

//
//  Items up for auction
//


class Item {
  std::string name;
  double reservedPrice;
  Time auctionEndsAt;

public:
  Item();
  Item (std::string itemName, double reserve, Time auctionEnd);

  // Access to attribute

  std::string getName() const {return name;}

  double getReservedPrice() const {return reservedPrice;}

  Time getAuctionEndTime() const {return auctionEndsAt;}
```

```
  /**
   * Read one item from the indicated file
   */
  void read (std::istream& in);

};


#endif
```

and

```
#include <iostream>
#include <fstream>

#include "items.h"

//
//  Items up for auction
//

using namespace std;




Item::Item()
  : reservedPrice(0.0)
{
}

Item::Item (std::string itemName, double reserve, Time auctionEnd)
  : name(itemName), reservedPrice(reserve), auctionEndsAt(auctionEnd)
{
}


/**
```

```
 * Read one item from the indicated file
 */
void Item::read (istream& in)
{
  in >> reservedPrice;
  auctionEndsAt.read (in);

  // Reading the item name.
  char c;
  in >> c; // Skips blanks and reads first character of the name
  string line;
  getline (in, line); // Read the rest of the line
  name = string(1,c) + line;
}
```

- The Time ADT:

```
#ifndef TIMES_H
#define TIMES_H

#include <iostream>

/**
 * Times in this program are represented by three integers: H, M, & S, representing
 * the hours, minutes, and seconds, respecitvely.
 */


struct Time {
  // Create time objects
  Time(); // midnight
  Time (int h, int m, int s);

  // Access to attributes
  int getHours() const;
  int getMinutes() const;
  int getSeconds() const;
```

```
  // Calculations with time
  void add (Time delta);
  Time difference (Time fromTime);

  /**
   * Read a time (in the format HH:MM:SS) after skipping any
   * prior whitepace.
   */
  void read (std::istream& in);


  /**
   * Print a time in the format HH:MM:SS (two digits each)
   */
  void print (std::ostream& out) const;

  /**
   *  Compare two times. Return true iff time1 is earlier than or equal to
   *  time2
   */
  bool noLaterThan(const Time& time2);


  /**
   *  Compare two times. Return true iff time1 is equal to
   *  time2
   *
   */
  bool equalTo(const Time& time2);

  // From here on is hidden
  int secondsSinceMidnight;
};

#endif // TIMES_H
```

and

```
#include "time.h"
#include <iomanip>
```

```
using namespace std;

/**
 * Times in this program are represented by three integers: H, M, & S, representing
 * the hours, minutes, and seconds, respecitvely.
 */



  // Create time objects
Time::Time() // midnight
{
  secondsSinceMidnight = 0;
}


Time::Time (int h, int m, int s)
{
  secondsSinceMidnight = s + 60 * m + 3600*h;
}

  // Access to attributes
int Time::getHours() const
{
  return secondsSinceMidnight / 3600;
}

int Time::getMinutes() const
{
  return (secondsSinceMidnight % 3600) / 60;
}

int Time::getSeconds() const
{
  return secondsSinceMidnight % 60;
}

// Calculations with time
void Time::add (Time delta)
```

```
{
  secondsSinceMidnight += delta.secondsSinceMidnight;
}


Time Time::difference (Time fromTime)
{
  Time diff;
  diff.secondsSinceMidnight =
    secondsSinceMidnight - fromTime.secondsSinceMidnight;
}


/**
 * Read a time (in the format HH:MM:SS) after skipping any
 * prior whitepace.
 */
void Time::read (std::istream& in)
{
  char c;
  int hours, minutes, seconds;
  in >> hours >> c >> minutes >> c >> seconds;
  Time t (hours, minutes, seconds);
  secondsSinceMidnight = t.secondsSinceMidnight;
}


/**
 * Print a time in the format HH:MM:SS (two digits each)
 */
void Time::print (std::ostream& out) const
{
  out << setfill('0') << setw(2) << getHours() << ':'
      << setfill('0') << setw(2) << getMinutes() << ':'
      << setfill('0') << setw(2) << getSeconds();
}



/**
```


```
 *  Compare two times. Return true iff time1 is earlier than or equal to
 *  time2
 */
bool Time::noLaterThan(const Time& time2)
{
  return secondsSinceMidnight <= time2.secondsSinceMidnight;
}



/**
 *  Compare two times. Return true iff time1 is equal to
 *  time2
 *
 */
bool Time::equalTo(const Time& time2)
{
  return secondsSinceMidnight == time2.secondsSinceMidnight;
}
```

- The BidCollection ADT:

```
#ifndef BIDCOLLECTION_H
#define BIDCOLLECTION_H

#include "bids.h"

class BidCollection {

  int MaxSize;
  int size;
  Bid* elements; // array of bids

public:

  /**
   * Create a collection capable of holding the indicated number of bids
   */
  BidCollection (int MaxBids = 1000);
```

```
  ~BidCollection ();



  // Access to attributes
  int getMaxSize() const {return MaxSize;}

  int getSize() const {return size;}



  // Access to individual elements

  const Bid& get(int index) const {return elements[index];}



  // Collection operations


  void addInTimeOrder (const Bid& value);
  //  Adds this bid into a position such that
  //   all bids are ordered by the time the bid was placed
  //Pre: getSize() < getMaxSize()



  void remove (int index);
  // Remove the bid at the indicated position
  //Pre: 0 <= index < getSize()


  /**
   * Read all bids from the indicated file
   */
  void readBids (std::string fileName);

};
```

```
#endif
```

and

```
#include "bidcollection.h"
#include "arrayUtils.h"
#include <fstream>

using namespace std;


/**
 * Create a collection capable of holding the indicated number of bids
 */
BidCollection::BidCollection (int MaxBids)
  : MaxSize(MaxBids), size(0)
{
  elements = new Bid [MaxSize];
}

BidCollection::~BidCollection ()
{
  delete [] elements;
}



// Collection operations


void BidCollection::addInTimeOrder (const Bid& value)
//  Adds this bid into a position such that
//   all bids are ordered by the time the bid was placed
//Pre: getSize() < getMaxSize()
{
  // Make room for the insertion
  int toBeMoved = size - 1;
  while (toBeMoved >= 0 &&
     value.getTimePlacedAt().noLaterThan(elements[toBeMoved].getTimePlacedAt())) {
    elements[toBeMoved+1] = elements[toBeMoved];
```


```
    --toBeMoved;
  }
  // Insert the new value
  elements[toBeMoved+1] = value;
  ++size;
}


void BidCollection::remove (int index)
// Remove the bid at the indicated position
//Pre: 0 <= index < getSize()
{
  removeElement (elements, size, index);
}

/**
 * Read all bids from the indicated file
 */
void BidCollection::readBids (std::string fileName)
{
  size = 0;
  ifstream in (fileName.c_str());
  int nBids;
  in >> nBids;
  for (int i = 0; i < nBids; ++i)
    {
      char c;

      string bidderName;
      double amount;
      in >> bidderName >> amount;

      Time bidPlacedAt;
      bidPlacedAt.read (in);

      string word, line;
      in >> word; // First word of itemName
      getline (in, line); // rest of item name

      string itemName = word + line;
```

```
      addInTimeOrder (Bid (bidderName, amount, itemName, bidPlacedAt));
    }
}
```

- The BidderCollection ADT:

```
#ifndef BIDDERCOLLECTION_H
#define BIDDERCOLLECTION_H

#include "bidders.h"

//
//  Bidders Registered for auction
//


class BidderCollection {

  int MaxSize;
  int size;
  Bidder* elements; // array of items

public:

  /**
   * Create a collection capable of holding the indicated number of items
   */
  BidderCollection (int MaxBidders = 1000);

  ~BidderCollection ();



  // Access to attributes
  int getMaxSize() const {return MaxSize;}

  int getSize() const {return size;}
```

```
  // Access to individual elements

  Bidder& get(int index) {return elements[index];}



  // Collection operations


  int add (const Bidder& value);
  //  Adds this bidder to the collection at an unspecified position.
  //  Returns the position where added.
  //Pre: getSize() < getMaxSize()



  void remove (int index);
  // Remove the item at the indicated position
  //Pre: 0 <= index < getSize()


  int findBidder (std::string name) const;
  // Returns the position where a bidde mathcing the given
  // name can be found, or getSize() if no bidder with that name exists.



  /**
   * Read all items from the indicated file
   */
  void readBidders (std::string fileName);

};


#endif
```

and

```
#include <iostream>
#include "arrayUtils.h"
#include <fstream>

#include "biddercollection.h"

using namespace std;


/**
 * Create a collection capable of holding the indicated number of items
 */
BidderCollection::BidderCollection (int MaxBidders)
  : MaxSize(MaxBidders), size(0)
{
  elements = new Bidder [MaxSize];
}

BidderCollection::~BidderCollection ()
{
  delete [] elements;
}



// Collection operations


int BidderCollection::add (const Bidder& value)
//  Adds this bidder
//Pre: getSize() < getMaxSize()
{
  addToEnd (elements, size, value);
  return size - 1;
}


void BidderCollection::remove (int index)
```

```
// Remove the bidder at the indicated position
//Pre: 0 <= index < getSize()
{
  removeElement (elements, size, index);
}




/**
 * Read all bidders from the indicated file
 */
void BidderCollection::readBidders (std::string fileName)
{
    size = 0;
    ifstream in (fileName.c_str());
    int nBidders;
    in >> nBidders;
    for (int i = 0; i < nBidders && i < MaxSize; ++i)
    {
      string nme;
      double bal;
      in >> nme >> bal;
      Bidder bidder (nme, bal);;
      add (bidder);
    }
}


/**
 * Find the index of the bidder with the given name. If no such bidder exists,
 * return nBidders.
 */
int BidderCollection::findBidder (std::string name) const
{
  int found = size;
  for (int i = 0; i < size && found == size; ++i)
    {
      if (name == elements[i].getName())
    found = i;
```

```
    }
  return found;
}
```

- The ItemCollection ADT:

```
#ifndef ITEMCOLLECTION_H
#define ITEMCOLLECTION_H

#include "items.h"

class ItemCollection {

  int MaxSize;
  int size;
  Item* elements; // array of items

public:

  /**
   * Create a collection capable of holding the indicated number of items
   */
  ItemCollection (int MaxItems = 1000);

  ~ItemCollection ();



  // Access to attributes
  int getMaxSize() const {return MaxSize;}

  int getSize() const {return size;}



  // Access to individual elements

  const Item& get(int index) const {return elements[index];}
```

```
  // Collection operations


  void addInTimeOrder (const Item& value);
  //  Adds this item into a position such that
  //   all items are ordered by the time at which the auction for the
  //   item ends.
  //Pre: getSize() < getMaxSize()



  void remove (int index);
  // Remove the item at the indicated position
  //Pre: 0 <= index < getSize()


  /**
   * Read all items from the indicated file
   */
  void readItems (std::string fileName);

};


#endif
```

and

```
#include <iostream>
#include "arrayUtils.h"
#include <fstream>

#include "itemcollection.h"

//
//  Items up for auction
//
```

```
using namespace std;


/**
 * Create a collection capable of holding the indicated number of items
 */
ItemCollection::ItemCollection (int MaxItems)
  : MaxSize(MaxItems), size(0)
{
  elements = new Item [MaxSize];
}

ItemCollection::~ItemCollection ()
{
  delete [] elements;
}



// Collection operations


void ItemCollection::addInTimeOrder (const Item& value)
//  Adds this item into a position such that
//   all items are ordered by the time the item' auction ends
//Pre: getSize() < getMaxSize()
{
  // Make room for the insertion
  int toBeMoved = size - 1;
  while (toBeMoved >= 0 &&
     value.getAuctionEndTime().noLaterThan(elements[toBeMoved].getAuctionEndTime())) {
    elements[toBeMoved+1] = elements[toBeMoved];
    --toBeMoved;
  }
  // Insert the new value
  elements[toBeMoved+1] = value;
  ++size;
}
```


```
void ItemCollection::remove (int index)
// Remove the item at the indicated position
//Pre: 0 <= index < getSize()
{
  removeElement (elements, size, index);
}



/**
 * Read all items from the indicated file
 */
void ItemCollection::readItems (std::string fileName)
{
  size = 0;
  ifstream in (fileName.c_str());
  int nItems;
  in >> nItems;
  for (int i = 0; i < nItems; ++i)
    {
      Item item;
      item.read (in);
      addInTimeOrder (item);
    }
}
```

Note how the vast majority of the code is now a collection of encapsulated ADTs, with only a limited amount of very application-specific code left outside.

This is very typical of well-written C++ applications.

. . . . . .