Debugging

Steven Zeil

July 17, 2013

Contents

Testing versus Debugging

- *Testing* is the act of executing a program with selected data to uncover bugs.
- *Debugging*, is the process of finding the faulty code responsible for failed tests.

. . . . . .

Rules of Debugging

1. You can't debug what you don't understand
2. Make it fail...
   (a) Make it fail *reliably*
   (b) Make it fail *easily*
   (c) Make it fail *visibly*
3. Track down the culprit
   (a) Don't Guess, *Hypothesize*
   (b) Divide and Conquer
   (c) If you didn't fix it, it ain't fixed!

- Today we'll focus on the first two
- See also *Debugging* by David Agans, and www.debuggingrules.com

. . . . . .

1 You can't debug what you don't understand

You can't debug what you don't understand

How can you hope to fix (or even recognize) what is wrong if you don't know what it will look like when it's right?

- How can you even test it in the first place?

. . . . . .

Read the Documentation

- Read all available documentation before you start testing
- Then read it again when you start debugging
  - Most common advice on any technical help site: RTFM

. . . . . .

Know What's Reasonable

- Technical knowledge:
  - what are common values in various data types?
  - how does this algorithm normally behave?
- Domain & application knowledge:
  - What will correct output look like?
  - What are reasonable ranges for values?

. . . . . .

Know the Roadmap

- What are the pieces of the system?
- How do they connect/interact with one another?

. . . . . .

2 Make it fail. . .

Make it fail. . .

Typically you start debugging because you program failed

- maybe during your own testing
- or someone else told you about it

What's the next thing you need to do?

. . . . . .

Make It Fail Again

Once You've Made it Fail. . .

- Make sure you can make it fail again
- This is where testing and debugging meet

. . . . . .

Why "Make It Fail Again"?

- You need to observe the failure
  - Second hand reports and even your own memories won't be detailed enough
- You need to figure out what's causing it
  - Knowing the conditions under which it fails is an important clue
- You need to know if you have fixed it
  - If you know how to cause the failure, then apply a fix, you can see if the failure really goes away.

. . . . . .

2.1 Make it fail Reliably

Make it fail Reliably

- Try to repeat whatever you did to cause the failure
- Write down your steps
  - Try to get to the point where you can automatically produce the failure.
    - For programs with textual input, if the failure is replicatable, you should be able to do this. E.g.,

      ```
      program < inputData.txt
      ```

    - This is a good reason not to enter your tests manually in the first place.
- Sometimes, replicating the failure is hard. . .

. . . . . .


Intermittent Failures

- A failure that, given the same inputs, occurs only some of the time is called *intermittent*
  - These are the bane of technicians and engineers in many fields
- An intermittent failure may occur 99% of the time when the buggy input is presented, or only 1% of the time, or even less.
  - Can waste a lot of time during debugging waiting for a failure

. . . . . .

Can Hardware Failures Be Truly Intermittent?

- Sometimes we just don't see the common factor that causes the failure
  - So we just "think" the problem is intermittent.
- But hardware is subject to "random" factors such as vibration, electronic noise, loose connections, temperature variations, timing variations
  - People working with hardware need to be creative to control or reproduce these factors.

. . . . . .

Can Software Failures Be Truly Intermittent?

- Sometimes
  - software that controls hardware can have timing issues
  - So can software that runs on distributed hardware
- But usually we are simply not seeing the critical factor

. . . . . .

Tracking Down Intermittent Failures

- Work backwards - the problem may be in an earlier input sequence
- Run repeatedly (invoke the buggy code from within a loop)
  - add code to detect the failure automatically
- Take advantage of the nature of the failure
  - Focus on the most likely causes of intermittent failures

. . . . . .

What Causes Intermittent Software Failures?

Typically,

- Uninitialized variables
- Out of bounds data access
- Pointer errors

. . . . . .

Uninitialized variables

- Errors are more likely to be seen in longer test sequences
- Often easy to catch "by eye"
  - Look for bizzare, apparently random numbers in the output
- Manifests more often in Linux than in Windows?
  - Windows tends to zero out newly allocated memory

  - Because zero is the most common thing we would have *wanted* to use to initialize, this "hides" the problem.
    * Until the most embarrassing moment possible

. . . . . .

Out of bounds data access

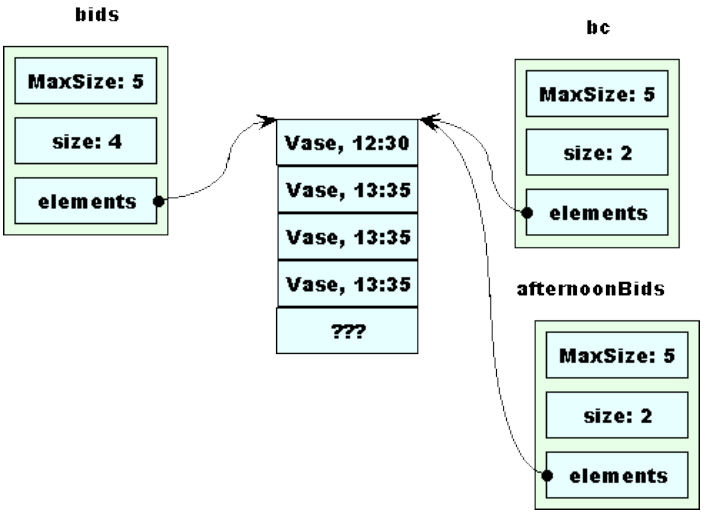
- What happens if you access arrays using negative or overly large index values?
  - May retrieve data that's actually not in the array, but part of some other variable
  - If you store data via the array, you may actually be overwriting other variables
- Timing and actual symptom of failure may depend on how/when those other variables get used.

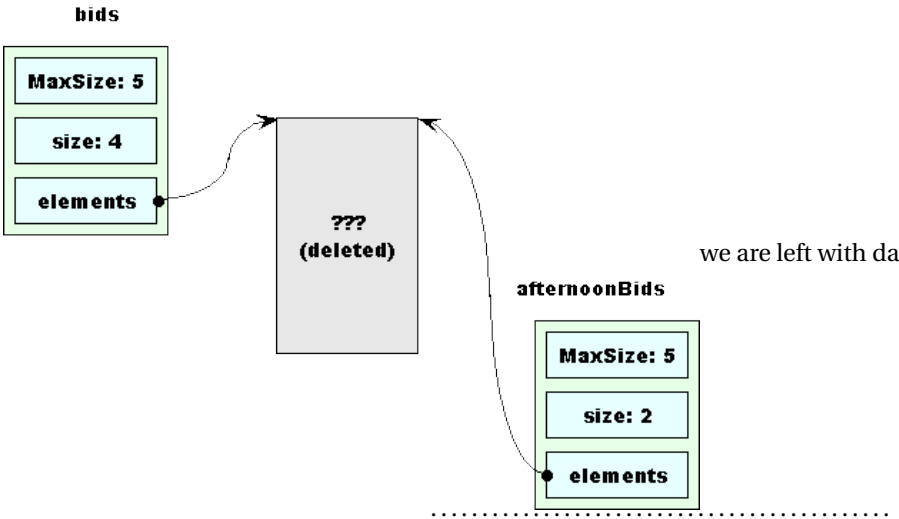
. . . . . .

Pointer errors - dangling pointers

Dangling pointers

We saw earlier that if bc is destroyed,...

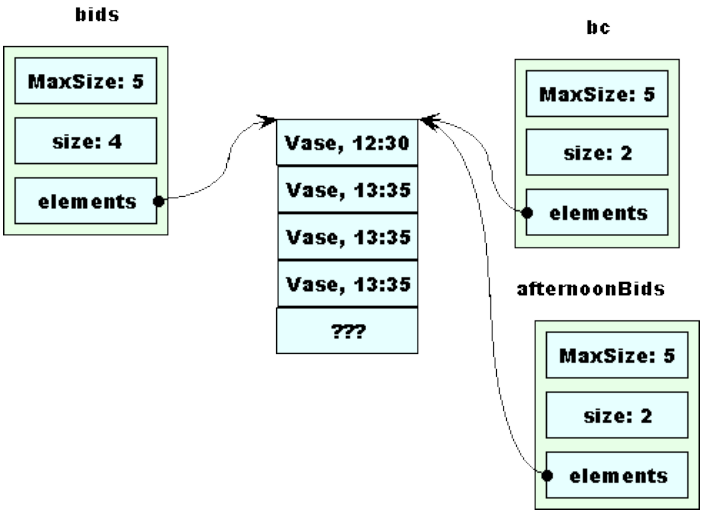




we are left with dangling pointers.

Pointer errors - Deleting a pointer twice

What happens if all the collections were destroyed and they each deleted the pointer?



. . . . . .

2.2 Make it fail Easily

Make it fail Easily

Debugging often requires you to run the failing test cases over and over again.

- If it takes hours to demonstrate a failure, how long will it take you to debug it?
- Even if it merely takes several minutes, how often are you going to run that failing code?

. . . . . .

Simplify Your Failing Tests

- Debugging is primarily an activity of working backwards from the point of failure towards the cause

– The shorter the path from the start of execution to the failure, the less you need to look at.

Simplify, Simplify, Simplify

If you made the program fail on the 100th input or 100th time around a loop, can you find a simpler test that fails after the 10th time?

"The only reasonable numbers in a programming language design are zero, one and infinity." – Bruce MacLennan (1987)

- Or maybe the 2nd?

KISS

Later we will make a distinction between unit testing and system testing,

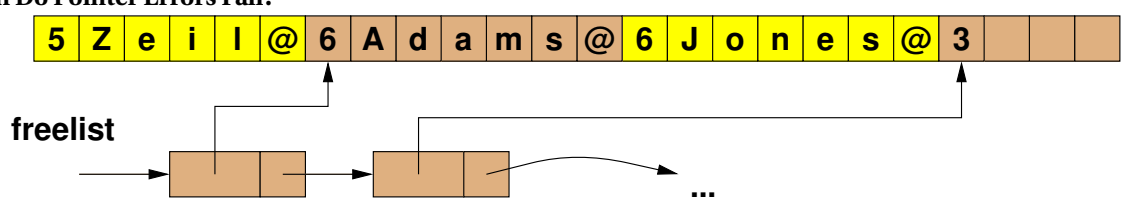
"Two is an impossible number, and can't exist." – I. Asimov (1972)

- Unit tests are typically much simpler
- One of the major reasons to use them

If It Takes Too Long to Fail

- Redefine failure!
  - By adding more and *earlier* output, you can move up the point of failure and see things going wrong sooner.

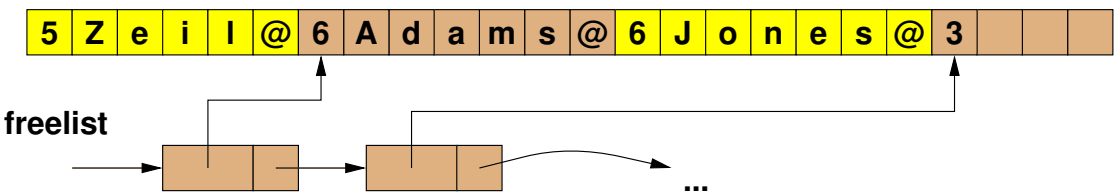
Example: When Do Pointer Errors Fail?



- When memory is deleted, it usually gets added to a "free list"
- If we try to access a freed block via a *dangling pointer*
  - If the block has not yet been reallocated for a new purpose, everything still looks OK
  - Possibly much later in the program, when that block is re-used, we suddenly start seeing problems.

. . . . . .

Example: When Do Pointer Errors Fail?



- If we delete the same block twice...
  - Tends to corrupt the free list
    - Probably won't fail immediately
  - Later in the execution, may suddenly be impossible to allocate new memory
  - Or may eventually allocate same block twice for two different variables

. . . . . .

Tracking Pointer Errors - Redefining Failure

- That time delay between the problem and the actual failure can be hard to deal with
- If we can reduce the delay, error will no longer appear intermittent

  Possibilities:

  - Print addresses being allocated and deleted
  - Add debugging output to constructors and destructors
  - Specialized tools (e.g., Purify, LeakTracer)

- Idea is to make the failure *obvious* and *early*
  - Related to the next rule, ...

. . . . . .

2.3 Make it fail Visibly

Make it fail Visibly

- If a tree falls in the forest and no one is there to hear, does it make a sound?
- If a program fails and no one observes the error, can you debug it?

. . . . . .

Program States

A *program state* is the combination of

- what you've got (the values of all program variables)
- where you are (the location to which execution has reached)
- how you got here (the activation stack recording the places from which all still-active function calls were made)

. . . . . .

Infected States

A program state is *infected* if one of its components is affected by a bug

- bad value in one or more variables, or
- at a wrong location (from following an incorrect branch)

. . . . . .

Propagating an Infection

As we move forward through the code, an infected state can

- spread - as bad values in some variables are used to compute the values of other variables, or
- vanish - if the infected values are never written to output and/or are overwritten by good values

. . . . . .

From Infection to Failure

A failure occurs when an infected state affects the program output.

- Debugging is a process of tracing backwards from bad output towards the cause (faulty code)
- Requires viewing of the infected states

. . . . . .

2.3.1 Instrument the System

Instrument the System

- Add debugging output
  - Tells you where you are

  - Display the value of variables suspected of being infected

- Usually written to the standard error stream (*cerr*)
  - Minimizes interference with "real" output written to standard out (*cout*)
  - Can be separated out at the command line.
    On our Unix system (running tcsh):
    * `myProgram.exe`
      writes *cout* and *cerr* to screen
    * `myProgram.exe > output.dat`
      writes *cout* to file `output.dat` and *cerr* to screen
    * `myProgram.exe >& output.dat`
      writes both *cout* and *cerr* to file `output.dat`

. . . . . .

Design Instrumentation In

Good rule of thumb for C++ ADT designers:

Always provide an output function for your ADT, showing the value of each data member.

- Even if the application itself does not need it
- More often than not, you will want it during debugging
  - And you don't want to be distracted by designing, writing and debugging new output functions when you are already debugging something else.

. . . . . .

Build Instrumentation In Later

- Actual outputs are likely to be added during debugging

```
cerr << "I suspect " << x
   << " is wrong" << endl;
```

- Special macros useful during debugging:
  - `__FILE__` expands to the name of the file being compiled
  - `__LINE__` expands to the line number in which it appears

```
cerr << "I suspect " << x
   << " is wrong at "
   << __FILE__ << ":" << __LINE__<< endl;
```

- You can even make your own macros:

```
#define DEBUGOUT(x) cerr << #x << "@" \
   << __File__ << ":" << __LINE__ << " " << x << endl;
```

  - Try running

```
#include <iostream>

using namespace std;


#define DEBUGOUT(x) cerr << #x << "@" << __FILE__ << ":" << __LINE__
   << ": " << x << endl;


int main()
```


```
{
  int k = 0;
  string y = "hello";
  cout << "Real output " << k << endl;
  cerr << "Debugging output " << k+1 << endl;
  cout << "Real output " << y << endl;
  cerr << "I suspect " << y << " is incorrect at "
       << __FILE__ << ":" << __LINE__ << endl;
  cout << "Real output " << k << endl;
  DEBUGOUT (k);
  DEBUGOUT (y);
  DEBUGOUT (k + k);
  DEBUGOUT (y.size());
  return 0;
}
```

.....

Never Throw a Debugging Tool Away

When you have finished debugging a problem, deactivate your debug code, don't delete it!

- You may need it again for the next bug.
- Different ways to deactivate
  - Comment it out

```
  x = foo(x,y);
  cerr << "x is " << x << endl;
  x = bar(x,y,z);
```

becomes

```
  x = foo(x,y);
  // cerr << "x is " << x << endl;
  x = bar(x,y,z);
```

– Use ifdefs:

```
  x = foo(x,y);
#ifdef DEBUG
  cerr << "x is " << x << endl;
#endif
  x = bar(x,y,z);
```

or

```
  x = foo(x,y);
#ifndef NDEBUG
  cerr << "x is " << x << endl;
#endif
  x = bar(x,y,z);
```

– Build deactivation in to debugging macros

```
#ifdef DEBUG
#define DEBUGOUT(x) cerr << #x << "@" \
   << __File__ << ":" << __LINE__ << " " << x << endl;
#else
#define DEBUGOUT(x) // ignored if not debugging
#endif
```

*
```
#include <iostream>

using namespace std;


#ifdef DEBUG
```


```
#define DEBUGOUT(x) cerr << #x << "@" << __FILE__ << ":" << __LINE__
   << ": " << x << endl;
#else
#define DEBUGOUT(x) // ignored if not debugging
#endif

int main()
{
  int k = 0;
  string y = "hello";
  cout << "Real output " << k << endl;
#ifdef DEBUG
  cerr << "Debugging output " << k+1 << endl;
#endif
  cout << "Real output " << y << endl;
#ifndef NDEBUG
  cerr << "I suspect " << y << " is incorrect at "
       << __FILE__ << ":" << __LINE__ << endl;
#endif
  cout << "Real output " << k << endl;
  DEBUGOUT (k);
  DEBUGOUT (y);
  DEBUGOUT (k + k);
  DEBUGOUT (y.size());
  return 0;
}
```

.....

The Heisenberg Uncertainty Principle

Debugging instrumentation may affect the behavior of the bug

- Sometimes unavoidable (but that's pretty rare)
- Be careful
  - After adding instrumentation, check to be sure you can still reproduce the bug

. . . . . .

3 Example: the Auction Program

Example: the Auction Program

As last seen: Auction ADTs (??)

. . . . . .

Failure Report

During an early run, the seller of a Chinese Vase complained that a bid for $27 was accepted as a winner even though a reserve price of $50 had been set.

- A check of the items file:

```
24
10.00  12:00:00  Star Wars Collectables
1.00   23:00:00  Bonds autograph
  ...
50.00  20:06:27  Chinese Vase
  ...
100.00   23:30:00  Cornflake shaped like Illinois
```

  showed that the reserve price was indeed set.

- The bids file did indeed contain a bid for the indicated amount.

```
4000
  ⋮
jjones 26.00 00:18:03 Hummel Figurine
  ⋮
ssmith 27.00 04:03:05 Chinese Vase
  ⋮
```

- Rerunning the program showed that the failure was reliable.

. . . . . .

3.1 Simplifying the Failing Tests

Simplifying the Failing Tests

- If the program was run with *only* the $27 bid, it correctly rejects that bid and announces that the reserve price on the vase was not met.
- We can easily select only the bids on the vase:

  ```
  grep "Chinese Vase" items.dat > items0.dat
  ```

  but the failure does not then occur. Instead, a $60 bid by jjones is selected as the winner.

. . . . . .

Rounding Up the Suspects

- A closer look at `items0.dat` shows that only two people, jjones and ssmith, bid on the vase.
- We can collect their bids for the day

  ```
  grep "jjones" bids.dat > bids0a.dat
  grep "ssmith" bids.dat > bids0b.dat
  cat bids0a.dat bids0b.dat > bids0.dat
  ```


and discover that they bid on only two items.

- Collecting those two items:

```
2
50.00 20:06:27 Chinese Vase
25.00 15:30:11 Hummel Figurine
```

those two bidders:

```
2
jjones 75.00
ssmith 55.00
```

and the related bids:

```
4
jjones 25.00 05:00:00 Chinese Vase
jjones 26.00 00:18:03 Hummel Figurine
jjones 60.00 04:03:01 Chinese Vase
ssmith 27.00 04:03:05 Chinese Vase
```

we are able to reproduce the failure reliably:

```
Hummel Figurine won by jjones for 26
Chinese Vase won by ssmith for 27
```

. . . . . .

3.2 Instrument the Code

Instrument the Code

The most obvious place to check for this problem is in the `resolveAuction` function

- That's where the incorrect output is produced
  - So we can work backwards from there

- Should we first check the input to see if everything is being read in correctly?
  - Probably not an effective way to use our time
  - Most of the program is probably correct (have already passed many tests)
    - So we could waste a lot of time checking everything that works
    - Exception would be if this were a divide and conquer strategy (later)

.....

3.3 Working Backwards

Working Backwards

Problem appears to be in the selection of `winningBidderSoFar` and `highestBidSoFar`.

- i.e., these are infected

```
/**
 * 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;
      }
}
```

These are initialized outside (before) the loop and reset inside, so we might want to check that the values are reasonable when they are reset.

.....

When a New Best Bid is Found

```
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();
                cerr << "Best bid so far is "
          << highestBidSoFar << " from "
                       << winningBidderSoFar << endl;
              }
          }
          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;
      }
}
```

gives us the output:

```
Best bid so far is 26 from jjones
Hummel Figurine won by jjones for 26
Best bid so far is 27 from ssmith
Chinese Vase won by ssmith for 27
```

which does not really tell us much.

One problem is that there's just not enough detail.

.....

Adding Detail

```
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();
                 >cerr << "Best bid so far is ["
          << bid.getBidder() << " "
          << bid.getAmount() << " "
          << bid.getItem() << " ";
      bid.getTimePlacedAt().print(cerr);
      cerr << "] from ["
          << bidder.getName() << " "
          << bidder.getBalance() << "]"
          << endl;
              }
          }
          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;
      }
}
```

gives us the output:

```
Best bid so far is [jjones 26 Hummel Figurine 00:18:03] from [jjones 75]
Hummel Figurine won by jjones for 26
Best bid so far is [ssmith 27 Chinese Vase 04:03:05] from [ssmith 55]
Chinese Vase won by ssmith for 27
```

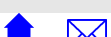
all of which looks OK.

- Actually I would never write anything as clumsy as that
- Compare the effort required to print all the fields of a bid or bidder to the effort required to print all the fields of a time
  - That's because, for `Time`, we observed the rule of thumb to actually provide an output function.

.....

Adding Detail the Easy Way

```
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();
                   cerr << "Best bid so far is ";
        bid.print (cerr);
        cerr << " from ";
        bidder.print (cerr);
        cerr << endl;
                }
            }
            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;
      }
}
```

For this to work, though, we need to add the `print` functions to bids and bidders (that we should probably have had there in the first place.

.....

Auction with Output Functions

- The BidCollection ADT:

```
#ifndef BIDCOLLECTION_H
#define BIDCOLLECTION_H

#include "bids.h"
#include <iostream>

class BidCollection {

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

public:

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

  ~BidCollection ();



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

  int getSize() const {return size;}



  // Access to individual elements
```

.....

Looking at All Bids

Since the determination of the best bid is not obviously wrong, we might wonder if all the non-best bids are being handled correctly. In particular, we recall that there were more than two bids for the two items.

```
void resolveAuction (const Item& item,
                     BidderCollection& bidders,
                     BidCollection& bids)
{
  cerr << "resolving item ";
  item.print(cerr);
  cerr << endl;
  double highestBidSoFar = 0.0;
  string winningBidderSoFar;
  bool reservePriceMet = false;
  for (int bidNum = 0; bidNum < bids.getSize(); ++bidNum)
    {
      Bid bid = bids.get(bidNum);
      cerr << "Looking at bid ";
      bid.print (cerr);
      cerr << endl;
      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);
```



```
          cerr << "Bidder is ";
          bidder.print (cerr);
          cerr << endl;
              // Can this bidder afford it?
              if (bid.getAmount() <= bidder.getBalance())
                {
                  highestBidSoFar = bid.getAmount();
                  winningBidderSoFar = bid.getBidder();
      cerr << "Best bid so far is ";
      bid.print (cerr);
      cerr << " from ";
      bidder.print (cerr);
      cerr << endl;
                }
           }
           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;
      }
}
```

gives output

```
resolving item [Hummel Figurine 25 15:30:11]
Looking at bid [jjones 26 Hummel Figurine 00:18:03]
Bidder is [jjones 75]
Best bid so far is [jjones 26 Hummel Figurine 00:18:03] from [jjones 75]
Looking at bid [jjones 60 Chinese Vase 04:03:01]
Looking at bid [ssmith 27 Chinese Vase 04:03:05]
Looking at bid [jjones 25 Chinese Vase 05:00:00]
Hummel Figurine won by jjones for 26
resolving item [Chinese Vase 50 20:06:27]
Looking at bid [jjones 26 Hummel Figurine 00:18:03]
Looking at bid [jjones 60 Chinese Vase 04:03:01]
Bidder is [jjones 49]
Looking at bid [ssmith 27 Chinese Vase 04:03:05]
Bidder is [ssmith 55]
Best bid so far is [ssmith 27 Chinese Vase 04:03:05] from [ssmith 55]
Looking at bid [jjones 25 Chinese Vase 05:00:00]
Chinese Vase won by ssmith for 27
```

It's still not clear why the low bid by smith is being accepted, but we can see that all bids are in fact being processed in some fashion.

Since we are not getting enough info about the reserve, let's also monitor when the *reservePriceMet* flag changes.

. . . . . .

When Does reservePriceMet change?

```
void resolveAuction (const Item& item,
                     BidderCollection& bidders,
                     BidCollection& bids)
{
  cerr << "resolving item ";
  item.print(cerr);
  cerr << endl;
  double highestBidSoFar = 0.0;
  string winningBidderSoFar;
  bool reservePriceMet = false;
  for (int bidNum = 0; bidNum < bids.getSize(); ++bidNum)
    {
      Bid bid = bids.get(bidNum);
      cerr << "Looking at bid ";
      bid.print (cerr);
      cerr << endl;
      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);
          cerr << "Bidder is ";
          bidder.print (cerr);
          cerr << endl;
              // Can this bidder afford it?
              if (bid.getAmount() <= bidder.getBalance())
                {
                  highestBidSoFar = bid.getAmount();
```


```
                winningBidderSoFar = bid.getBidder();
      cerr << "Best bid so far is ";
      bid.print (cerr);
      cerr << " from ";
      bidder.print (cerr);
      cerr << endl;
               }
           }
           if (bid.getAmount() > item.getReservedPrice())
         {
               reservePriceMet = true;
               cerr << "reserve price met by bid ";
     bid.print (cerr);
     cerr << endl;
         }
       }
   }

   // 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;
      }
}
```

which has output

```
resolving item [Hummel Figurine 25 15:30:11]
Looking at bid [jjones 26 Hummel Figurine 00:18:03]
Bidder is [jjones 75]
Best bid so far is [jjones 26 Hummel Figurine 00:18:03] from [jjones 75]
reserve price met by bid [jjones 26 Hummel Figurine 00:18:03]
Looking at bid [jjones 60 Chinese Vase 04:03:01]
reserve price met by bid [jjones 60 Chinese Vase 04:03:01]
Looking at bid [ssmith 27 Chinese Vase 04:03:05]
reserve price met by bid [ssmith 27 Chinese Vase 04:03:05]
Looking at bid [jjones 25 Chinese Vase 05:00:00]
Hummel Figurine won by jjones for 26
resolving item [Chinese Vase 50 20:06:27]
Looking at bid [jjones 26 Hummel Figurine 00:18:03]
Looking at bid [jjones 60 Chinese Vase 04:03:01]
Bidder is [jjones 49]
reserve price met by bid [jjones 60 Chinese Vase 04:03:01]
Looking at bid [ssmith 27 Chinese Vase 04:03:05]
Bidder is [ssmith 55]
Best bid so far is [ssmith 27 Chinese Vase 04:03:05] from [ssmith 55]
Looking at bid [jjones 25 Chinese Vase 05:00:00]
Chinese Vase won by ssmith for 27
```

.....

Diagnosis

Looking at that output:

```
resolving item [Hummel Figurine 25 15:30:11]
Looking at bid [jjones 26 Hummel Figurine 00:18:03]
Bidder is [jjones 75]
Best bid so far is [jjones 26 Hummel Figurine 00:18:03] from [jjones 75]
reserve price met by bid [jjones 26 Hummel Figurine 00:18:03]
Looking at bid [jjones 60 Chinese Vase 04:03:01]
reserve price met by bid [jjones 60 Chinese Vase 04:03:01]
Looking at bid [ssmith 27 Chinese Vase 04:03:05]
reserve price met by bid [ssmith 27 Chinese Vase 04:03:05] ❶
Looking at bid [jjones 25 Chinese Vase 05:00:00]
Hummel Figurine won by jjones for 26
resolving item [Chinese Vase 50 20:06:27]
Looking at bid [jjones 26 Hummel Figurine 00:18:03]
Looking at bid [jjones 60 Chinese Vase 04:03:01]
Bidder is [jjones 49]                                    ❸
reserve price met by bid [jjones 60 Chinese Vase 04:03:01] ❷
Looking at bid [ssmith 27 Chinese Vase 04:03:05]
Bidder is [ssmith 55]
Best bid so far is [ssmith 27 Chinese Vase 04:03:05] from [ssmith 55]
Looking at bid [jjones 25 Chinese Vase 05:00:00]
Chinese Vase won by ssmith for 27
```

Now we can see, not just one, but two problems.

❶ The Hummel reserve price is being met by a bid on the Chinese vase

- This did not affect the output because other bids also met the reserve.

❷ The Chinese vase reserve price is being met by a bid from someone who does not have enough money (❸) to pay that bid.

So the condition

```
if (bid.getAmount() > item.getReservedPrice())
```


```
        {
          reservePriceMet = true;
```

is far too lax.

. . . . . .

Diagnosis Precedes the Cure

At this point, we have diagnosed the problem.

- That's not the same thing as actually fixing it, but it's close.
- In fact, we mainly need to move that test inside the if statements above it. (auctionMain.cpp)

. . . . . .

4 Lessons

Lessons

- Simplify the failing test cases
- Add debugging output to cerr
- Plan for debugging by adding output functions to ADTs
- Work backwards from the point of failure
- Deactivate debugging output; don't throw it away

. . . . . .