Defensive Programming

Steven Zeil

July 22, 2013

Contents

Defensive Programming

Defensive programming is an approach to coding based on minimizing assumptions made by the programmer.

- Assumes that things *will* go wrong
- Tries to predict and allow for likely problems

. . . . . .

1 Common Assumptions

The Glass is Half-Full

Programmers are optimists by nature. We always believe that our programs are actually going to run "as soon as I fix this one little problem" – despite the fact that, time and time again, we are proven wrong.

. . . . . .

Programmers Like to Assume...

- Program input will be good
- Function parameters will be good
- Algorithms will behave as expected (no bugs)

. . . . . Odds are, if we weren't such blind optimists, we would never actually dare to write a single line of code.

What to do About Assumptions?

- Document them
- Guard them

. . . . . .

2 Documenting Assumptions

Preconditions

It's common for functions to only work under certain circumstances.

- A *precondition* is a condition that must be true before the function is called, if we expect the function to do anything reasonable.
  - Preconditions are an obligation on the caller, not on the function being called

. . . . . .

Documenting Preconditions

- Preconditions should be documented in the function declaration.
  - Have to be visible to callers

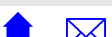
. . . . . .

Pre-Condition Example

Look at `addInTimeOrder` in

```
#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
```

.....

Internal Assumptions

Internal assumptions can be documented with comments.

E.g., in `resolveAuction`,

```
// If highestBidSoFar is non-zero, we have a winner
    if (reservePriceMet && highestBidSoFar > 0.0)
      {
        int bidderNum = bidders.findBidder(winningBidderSoFar);
        // winningBidderSoFar should be in the list of registered bidders
         cout << item.getName()
             << " won by " << winningBidderSoFar
```


```
            << " for " << highestBidSoFar << endl;
      Bidder& bidder = bidders.get(bidderNum);
      bidder.setBalance (bidder.getBalance() - highestBidSoFar);
    }
```

`bidderNum` should be valid, because all bids should be placed by bidders registered with the auction.
So we document this...

. . . . . .

3 Guarding Assumptions

Guarding Assumptions

Suppose that

- you have written a function that has a precondition
- you documented the precondition
- but your function is called with illegal parameters

It's not your fault, but what do you do?

. . . . . .

Possible Reactions

- Abort the program with an error message
- Ignore it and let the program continue
- Quietly correct the input and let the program continue.
- Issue an error message, correct the input, and let the program continue.

. . . . . .

Example

E.g., for

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

What should we do if `add` is called on a full collection?

.....

Abort the program with an error message

```
int BidderCollection::add (const Bidder& value)
//  Adds this bidder
//Pre: getSize() < getMaxSize()
{
  if (size < MaxSize)
    {
     addToEnd (elements, size, value);
     return size - 1;
    }
  else
    {
     cerr << "BidderCollection::add - collection is full" << endl;
     exit(1);
    }
}
```

- OK (but we'll see an easier way to do this).

- Some people are appalled at the idea of a program aborting is "real use", but the alternatives are often worse.

. . . . . .

Ignore it and let the program continue

- Worst thing we can do
  - We *know* that things are going badly, but we're going to risk incorrect output, corrupted output files and databases, etc.

. . . . . .

Quietly correct the input and let the program continue

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

- Nearly as dangerous as just letting the program run.
  - Usually cannot reasonably guess whether a given correction is safe or not

. . . . . .

Issue an error message, correct the input, and let the program continue

```
int BidderCollection::add (const Bidder& value)
//  Adds this bidder
//Pre: getSize() < getMaxSize()
{
  if (size < MaxSize)
    {
     addToEnd (elements, size, value);
     return size - 1;
    }
  else
    {
      cerr << "BidderCollection::add - collection is full" << endl;
     elements[size-1] = value;
    }
}
```

- Occasionally the best choice.
  - Depends on if anyone is watching for such error messages

.....

3.1 Guarding Assumptions with Assertions

Guarding Assumptions with Assertions

`assert(c);` (from the header file `<cassert>`) tests to see if a boolean condition `c` is true. If not, it issues an error message and aborts the program.

- Assertions can be deactivated by compiling with the macro symbol `NDEBUG` defined.

.....

assert() Example

```
    if (reservePriceMet && highestBidSoFar > 0.0)
      {
        int bidderNum = bidders.findBidder(winningBidderSoFar);
        // winningBidderSoFar should be in the list of registered bidders
        assert (bidderNum >= 0 && bidderNum < bidders.getSize());
        cout << item.getName()
             << " won by " << winningBidderSoFar
             << " for " << highestBidSoFar << endl;
        Bidder& bidder = bidders.get(bidderNum);
        bidder.setBalance (bidder.getBalance() - highestBidSoFar);
      }
```

.....

Guarding Preconditions

Perhaps the most common use of assertions is in guarding pre-conditions.

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

#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()
{
  assert (size < MaxSize);
  addToEnd (elements, size, value);
  return size - 1;
}


void BidderCollection::remove (int index)
// Remove the bidder at the indicated position
//Pre: 0 <= index < getSize()
{
  assert (0 <= index && index < MaxSize);
```


```
  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;
}


// Print the collection
void BidderCollection::print (std::ostream& out) const
{
  out << size << "/" << MaxSize << "{";
  for (int i = 0; i < size; ++i)
    {
      out << "  ";
      elements[i].print (out);
      out << "\n";
    }
  out << "}";
}
```

- If a program aborts because of a guarded precondition in a function, it's not the function's fault
  - It indicates a failure in the calling code.