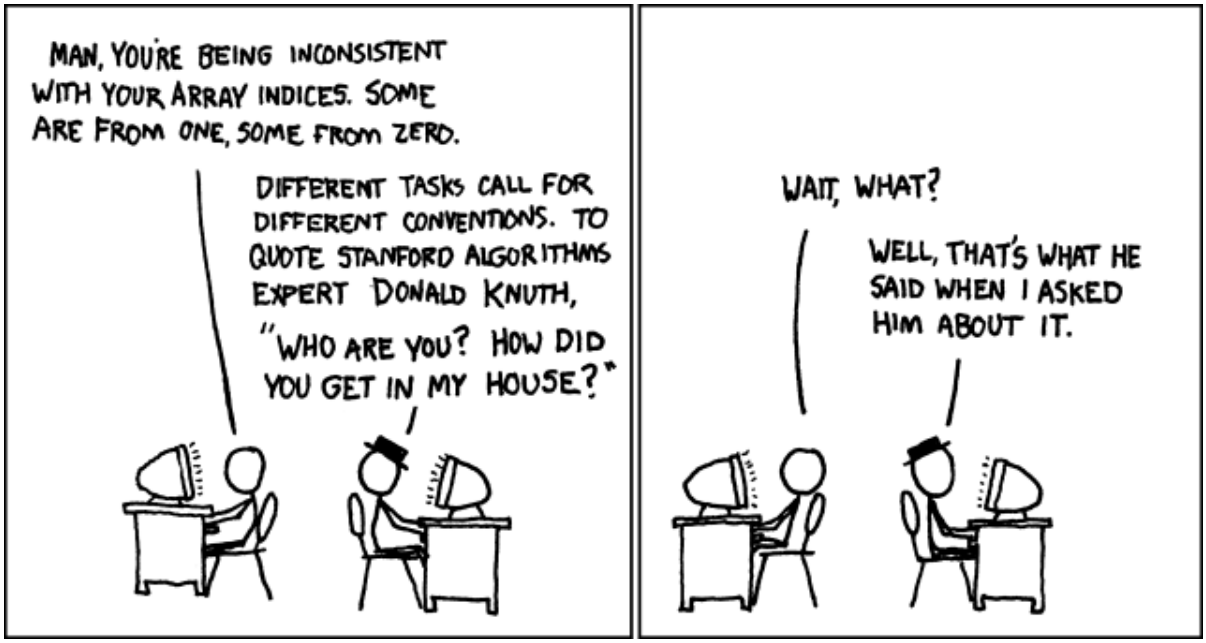
ARRAY OUT OF BOUNDS LAB



[1]

1. OBJECTIVE

In this lab, you will explore a common error in secure programming: array out of bounds errors. These errors can be easily overlooked and cause major problems in your program by overwriting other data. Furthermore, the lack of bounds checking on arrays can cause security holes through which hackers can insert malicious code into your program. We will go through a few simple examples of the ways that array out of bounds errors occur, and how to prevent them.

[1] Source: xkcd.com

2. INTRODUCTION

Imagine that you belong to a bank with safety deposit boxes. You own access to three safety deposit boxes that are part of a big line of boxes (like a series of small lockers, labeled A to Z). Say you own boxes D through F, in which you store valuables and cash. However, security in this bank is horrible! There is one key that accesses every box, so anyone can get into everyone else's boxes. What if the person who owns boxes A through C reached past his set of boxes into box D and took out your money? Or what if you accidentally went past box F into G to deposit your valuables? In both cases, you could lose what you thought you were keeping for safe storage!

The problem: array out of bounds

This safety deposit box analogy is similar to what can happen in computer memory when you try to access an index of an array that is not within the array's bounds. This common error is called an **array out of bounds** error, and it happens to even the most experienced programmers. The bad news is that it can cause serious problems by accessing information in another part of memory (when you look at the valuables in someone else's safety deposit box) or by overwriting the information that you have stored in your array (when someone else replaces the valuables in your box with their valuables). The good news is that this type of error is easy to avoid with some basic bounds checking.

Understanding the problem: how arrays work

When you declare an array in a program, you can think of it as reserving, or allocating, a series of boxes that are the size of the array's data type. Consider the following line of code:

```
int myNumbers[10];
```
[2]

[2] Note to the instructor: this lab focuses on stack arrays rather than heap arrays, since these errors are easier to recreate and control. If your students are familiar with the difference between stack and heap variables, you can explain that array out of bounds errors can also affect heap variables.

This allocates an array of 10 integer-sized boxes in memory where you can store 10 integer values.

Now, take:

```
char myLetters[10];
```

Here, you would still be able to store 10 character values, but the size of the array in memory is likely to be smaller. Characters are usually 1 byte while integers on most machines are 4 bytes.

To access an index of the array, you put the index that you want in the square brackets, e.g:

```
int numAccessed = myNumbers[2];
```

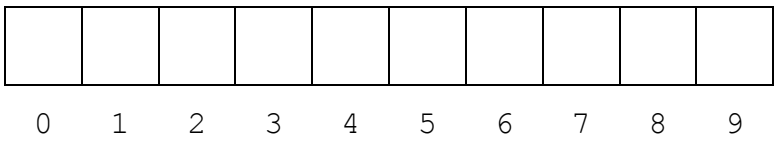
The most important thing to remember when considering array bounds is that arrays are zero-indexed[3]. This means that to access the first "box" in the array, you would want to say:

```
int firstNum = myNumbers[0];
```

Because of this, the last index in an array is the `(size-1)` index.

Consider the illustration below to better visualize this indexing:

`myNumbers`



What happens if you try to index `myNumbers[10]`? As you can see in the visual, 10 does not index into the `myNumbers` array. Instead, it will try to access the memory that is stored directly after the array on the stack. This could be another array, an instance variable, or anything else, but the important part is that it is not yours to mess with!

[3] In most languages, arrays are zero-indexed, though in languages such as SmallTalk, MATLAB, and Mathematica, they are indexed starting at one.

Common errors

One of the easiest ways to create an out of bound error is by looping through the indices of the array and going off one of the ends. In a `for` loop, it's fairly easy to catch -- knowing that you can only go to `size-1`, you can set the condition on the for loop to go up to `< size`. This will execute the loop at `size-1` but not at `size`. Consider the following `for` loop, which starts at the end of the array and goes backwards through each index `i` setting the i^{th} value in the array to `i`.

```
int size=5;
int arr[5];
for (int i = size; i > 0; i--){
     arr[i]=i;
}
```

This seems correct, since if you were to go from the beginning to end you would iterate from `i = 0` to `i < size`. However, notice that the `for` loop in the code above actually has two differences that may cause errors in your code:

1) It starts from `size`, and therefore accesses `arr[size]`, or `arr[5]` in this example, which is off the end of the array.
2) We never set `arr[0]`, since when we get to `i = 0`, the Boolean condition (`i > 0`) fails, and the `for` loop does not execute `arr[0]=0`.

There are a few ways to fix this, including starting from `size-1` and going to `i >= 0`, or starting from the beginning of the array and executing the `for` loop from `i = 0` to `i < size`.

A `while` loop can be even trickier if you are incrementing a counter, since it is up to you to decide when to check the condition and when to update your counter. Consider the following `while` loop:

```
int size = 5;
int i = 0;
int arr[5];
while (i < size) {
     i++;
```

```
        arr[i] = i;
    }
```

You might think that this does exactly the same thing as a for loop of the form

```
for(int i = 0; i < size; i++)
```

since you have a index "`i`", you check if it is less than "`size`", you increment it, and you set `arr[i] = i`. But if you look closely, you can see that you changed the order of checking the condition and incrementing `i`.

The errors that occur here are actually the same as those for the `for` loop above that starts at `i = size` and goes backwards through the array, even though we are going forward through the array in this example. These errors are:

1) The code doesn't set `arr[0] = 0`. We declare `int i = 0`, but before we set `arr[i] = i`, we increment `i`. Therefore we will have a garbage value in `arr[0]`. What would happen if you tried adding this `while` loop into your code?

2) When `i = size-1`, the code passes the `i<size` condition since `(size-1)<size`. Therefore, the code in the while loop is entered, incrementing `i` by I so that `i=size` and then tries to set `arr[size]=10`, which is out of bounds!

Another thing to consider is if you call a function to return an index of the array. This can happen if you have a function that takes in a key (such as a name or a number) and returns the index in the array that the key corresponds with.[4] Say that if an error occurs in your function (such as if the key does not have a valid index), the function returns `-1`, since this is not a valid index. If you don't check for this error return value, you might try to index into the -1 position of the array. Your function will try to store the data in the memory before the array, which can overwrite other important variables and cause major errors! An easy fix for this is to understand the functions that you call and what return values they might return, accounting for any error cases that can change what you want the function to do.

[4] Note to instructor: the most common use of this type of function is for a hash table. If your students know what hash functions are, you can point out to them that this is a common way to implement them.

3. BASIC EXERCISES

Let's consider a few basic examples that might help you understand where these errors occur and what problems they can cause.

3.1 ZEROING OUT AN ARRAY

Write a method that takes an array of `int`s and its size (an `int`) and uses a `for` loop to set each element within the array to be 0.

```
void ZeroArray(int[] array, int size) {









}
```

1. What happens to your program?

2. Insert "`cout << "i=" << i << endl;` " into your `for` loop. Assuming your function behaves correctly, what should your function print?

3. Could you replace a single character in your `for` loop condition to create an indexing error? What would that change be? What would be printed out in this case?

3.2 Bubble Sort

Bubble Sort is a somewhat simple linear sorting algorithm. It takes an array and repeatedly goes through it, comparing if an element is greater than the one after it, and swapping them if it is. The array is sorted when the algorithm runs through the whole array without making any swaps (i.e. when every element is less than the ones after it).

Let's consider an example:

We start with an initial array that we want to sort in *ascending* order using Bubble Sort.

We go through each neighboring pair of elements (the elements at positions 0 and 1, then the elements at positions 1 and 2, etc) in the array and swap the entries if they are in the wrong order. Note that if the two entries are the same value (6 and 6), we don't exchange them.

Now we repeat this process until we run through the entire array one full time without doing any swaps. At this point, we have sorted the array and we are done!

0	4	2	1	6	6

1st run through:

0	4	2	1	6	6	Note that the elements in positions 1 and 2 in the array (with values 4 and 2) are out of order, so they need to be swapped
0	2	4	1	6	6	Note that the elements in positions 2 and 3 (with values 4 and 1) are out of order, so they need to be swapped
0	2	1	4	6	6	The rest of the elements are in order

2nd run through:

0	2	1	4	6	6	Note that the elements in positions 1 and 2 (with values 2 and 1) are out of order, so they need to be swapped
0	1	2	4	6	6	The rest of the array elements are in order.

3rd run through:

0	2	1	4	6	6	All elements are in order, so no swapping is necessary

Now consider the following code:

```
#include <iostream>

using namespace std;

void bubbleSort(int arr[], int size);
void swap(int *a, int *b);
void printArray(string label, int arr[], int size);

int main() {
   const int size = 3;
   int arr[size] = {7, 1, 6};
   bubbleSort(arr, size);
   return 0;
}

void bubbleSort(int arr[], int size) {
   printArray("Original array", arr, size);

   while (true) {
      int swaps = 0;
      for (int n=0; n < size; n++) {
         if (arr[n] > arr[n+1]) {
            swap(&arr[n], &arr[n+1]);
            swaps++;
         }
      }
      if (swaps == 0) break;
   }

   printArray("Sorted array", arr, size);
}

void swap(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}

void printArray(string label, int arr[], int size) {
   cout << label << ":" << endl << "   ";
   string sep = "";
   for (int b=0; b<size; b++) {
      cout << sep << arr[b];
      sep = ", ";
   }
   cout << endl;
}
```

1. What do you get for your sorted array? Is that what you expected from the original?

2. What happened? (Hint: take a look at the `for` loop and consider the range of indices that it accesses in the array)

3. Can you fix the error so the correct array is sorted? (Hint: how can you change n to get the range of values that you want?)

4. BANKING FAILURES

Banks rely on computer software to keep track of all of the financial information for their clients. Occasionally you might hear in the news about a major bank failure or scandal due to a software error.

Consider the following error voted number 1 in the "Top 10 Software Failures of 2011" by Business Computing World:

> *"Financial services giant fined $25 million for hiding software glitch that cost investors $217 million*
>
> *A software error in the investment model used to manage client assets resulted in this international financial services giant being fined $25 million (£15.7 million) by the US Securities and Exchange Commission (SEC). The company also had to repay the $217 million (£136 million) backers lost when told that market volatility rather than software failure was to blame for their investment losses."* [5]

Usually, as with this case, the details of the software failure are not reported to the public. In this example, we will consider a simplified account system for a bank that could cause a major error like this one.

[5] Source: Codd, Phil. "Top 10 Software Failures of 2011." Business Computing World. 04 March 2013. <http://www.businesscomputingworld.co.uk/top-10-software-failures-of-2011/>.

To read more about this specific bank software error, refer to the article on Network World: http://www.networkworld.com/news/2011/020411-axa-rosenburg-group-glitch.html

4.1 EXAMPLE: ARRAY NATIONAL BANK

You are the IT expert for the Array National Bank. The person who held the job before you designed a very simple system for storing account balances: the bank has an array of balances, where each element in the array is the account balance of a specific customer. The bank also has an array of names (strings), each which correspond to the name of a bank customer. If a customer's name were in the 3rd index of the names array, that customer's account balance would be in the 3rd index of the balances array.

Your job is to implement the methods that your predecessor left blank. The skeleton code for the bank is as follows:

```
#include <iostream>
#include <sstream>

using namespace std;

/* This method should return the index of the given account name if it exists,
and -1 otherwise. */
int getAccountNum(string names[], string accountName, int numAccounts);


/* This method should print out the account balance of the given customer. */
void checkBalance(string names[], int accounts[], string accountName,
                  int numAccounts);

/* This method should effectively deposit the given amount into the given
   customer's account. */
void deposit(string names[], int accounts[], string accountName, int amount,
             int numAccounts);


/* This method should effectively withdraw the given amount from the given
   customer's account. */
void withdraw(string names[], int accounts[], string accountName, int amount,
              int numAccounts);


/* This method should print out all of the customers' names and account
   balances on separate lines. For example:

   John Doe: $120
   Fred Flintstone: $200
*/
void printAccounts(string names[], int accounts[], int numAccounts);

/* Do not worry about these two functions. */
void initAccounts(string names[], int accounts[], int numAccounts);
int getInt();
```

```
/* Do not alter this function. */
int main() {
   int numAccounts = 5;
   int accounts[numAccounts];
   string names[numAccounts];
   initAccounts(names, accounts, numAccounts);
   printAccounts(names, accounts, numAccounts);
   while (true) {
      cout << endl << "Enter customer name or press enter to quit: ";
      string accountName;
      getline(cin, accountName);
      if (accountName.empty()) break;
      cout << "Enter \"deposit\", \"withdraw\", "
           << "or press enter to check balance: ";
      string amountLine;
      getline(cin, amountLine);
      if (amountLine == "") {
         checkBalance(names, accounts, accountName, numAccounts);
      } else if (amountLine == "deposit") {
         cout << "Enter an amount (dollars) to deposit: ";
         int amount = getInt();
         deposit(names, accounts, accountName, amount, numAccounts);
         cout << "Successfully added $" << amount
              << " to the account of " << accountName << endl;
      } else if (amountLine == "withdraw") {
         cout << "Enter an amount (dollars) to withdraw: ";
         int amount = getInt();
         withdraw(names, accounts, accountName, amount, numAccounts);
         cout << "Successfully withdrew $" << amount
              << " from the account of " << accountName << endl;
      } else {
         cout << "Unrecognized command" << endl;
      }
   }
   return 0;
}


/* Do not alter these two functions. */
void initAccounts(string names[], int accounts[], int numAccounts) {
   for (int i = 0; i < numAccounts; i++) {
      accounts[i] = 100; // we have everyone start with $100
   }
   names[0] = "John Hancock";
   names[1] = "George Washington";
   names[2] = "Thomas Jefferson";
   names[3] = "Alexander Hamilton";
   names[4] = "Ben Franklin";
}

int getInt() {
   int amount;
   while (true) {
      string response;
      getline(cin, response);
      if (!(stringstream(response) >> amount)) {
         cout << "Enter a valid integer, try again: ";
      } else break;
   }
   return amount;
}
```

```
/* These are the functions you must implement. Descriptions of their behavior
 * are above
 */
int getAccountNum(string names[], string accountName, int numAccounts) {









}

void checkBalance(string names[], int accounts[], string accountName,
                  int numAccounts) {









}

void deposit(string names[], int accounts[], string accountName, int amount,
             int numAccounts) {









}

void withdraw(string names[], int accounts[], string accountName, int amount,
              int numAccounts) {









}
```

```
void printAccounts(string names[], int accounts[], int numAccounts) {




}
```

Review Your Code

When you start the program, you should get this output:

John Hancock: $100
George Washington: $100
Thomas Jefferson: $100
Alexander Hamilton: $100
Ben Franklin: $100

Enter customer name or press enter to quit:

1. First try entering "John Hancock" and press enter.
   Then a prompt "`Enter "deposit", "withdraw", or press enter to check balance:`" will appear. Type `deposit` and enter an amount to deposit into the account.

   Now say Benjamin Franklin comes in and wants to deposit $2000 (say he just made some money from his bifocal invention).
   Enter "Benjamin Franklin" as the customer name and 2000 as the amount to deposit.

   When Benjamin Franklin comes back the next day, he asks for you to check his account balance. This time he gives his name as "Ben Franklin" (he's more casual now that he knows you), so you enter this name. When you get the "deposit or withdraw" prompt, press enter. What number do you get as the account balance?

2. Is the account balance that you got expected? If so, how did you implement the methods in order to prevent array-out-of-bounds errors?
   If not, there may be an array-out-of-bounds error in your code. Where do you think this error is?
   (Hint: think about what account number "Benjamin Franklin" maps to)

Review a Coworker's Code

Your coworker has implemented the empty methods, as shown below:

```
int getAccountNum(string names[], string accountName, int numAccounts) {
    for (int i=0; i<numAccounts; i++) {
        if (names[i] == accountName) {
            return i;
        }
    }
    return -1; //not found
}


void checkBalance(string names[], int accounts[], string accountName, int numAccounts)
{
    int accountNum = getAccountNum(names, accountName, numAccounts);
    cout << accountName << "'s balance is $" << accounts[accountNum] << endl;
}


void adjustBalance(string names[], int accounts[], string accountName, int amount,
                    int numAccounts) {
    int accountNum = getAccountNum(names, accountName, numAccounts);
    accounts[accountNum]+=amount;
 }


void printAccounts(string names[], int accounts[], int numAccounts) {
    for (int i=0; i<numAccounts; i++) {
        cout << names[i] << " = $" << accounts[i] << endl;
    }
}
```

Replace your code with these methods and repeat the instructions from part 1 above. When you check the account balance of "Ben Franklin", what balance do you see?

There must be some sort of error. Try entering "Benjamin Franklin" for the name this time and check the account balance. What number do you get now?

Just for fun, enter your own name and check your account balance (it should be $0 since you aren't a customer at this bank). What do you get?

Let's figure out what happened.

1. Add the line `printAccounts(names, accounts, numAccounts);` to the end of the `while` loop in the main function (right after the "`cout << 'Successfully added...'`" line) so that the summary information prints after every change.
   Rerun the program repeating the steps above.

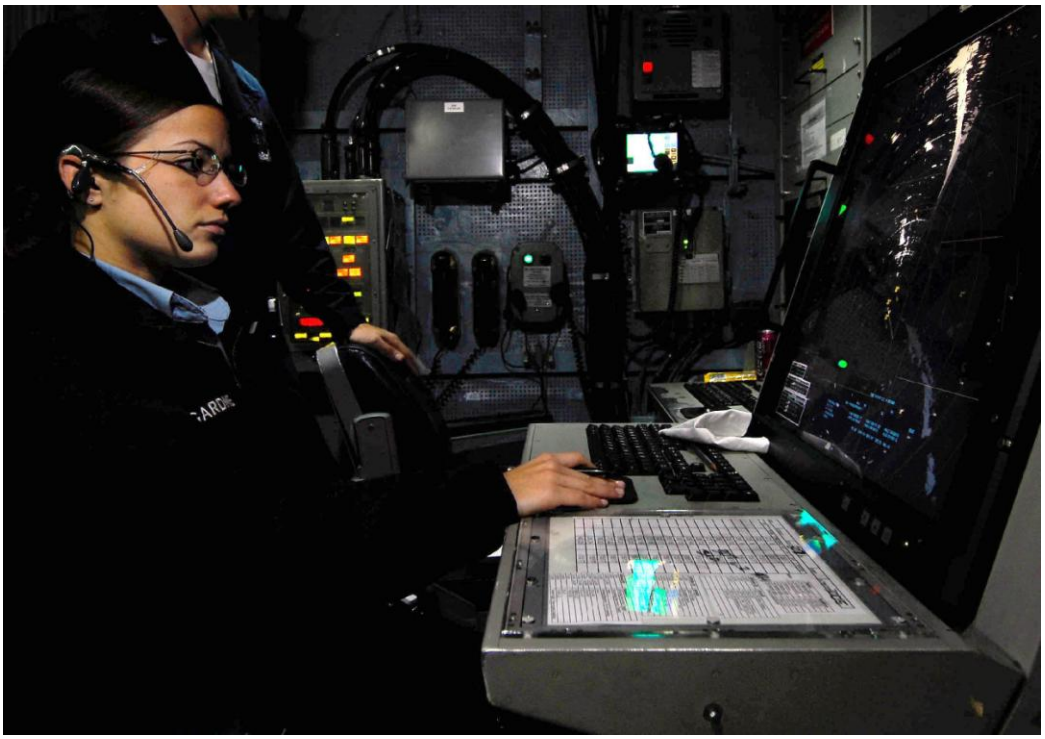
   After each entry, check the printed results. Do they reflect the change you made?
   If not, carefully consider why this occurred.
   (Hint: think about what account number Benjamin Franklin maps to).

2. Think about how you can change the code so that you don't make this error. What do you want to check for?

3. Make your changes and repeat the scenario from above. What results do you get now?

You can see how a simple problem like this could cause mayhem in a bank if it went unnoticed for a long time. Customers might think that they deposited money into their account, only to come back to check the balance and find that it is not there! Using array bounds checking can avoid errors such as this one.

5. Los Angeles Air Traffic Control Software Failure

 [6]

It is September 14th, 2004, and you are waiting at the Los Angeles International Airport for your flight. Suddenly, someone announces that all flights in and out of the airport have been cancelled for the night! This is what happened when the Los Angeles air traffic control center unexpectedly lost voice contact with 400 planes it was tracking in the southwestern United States.

A software failure on the Voice Switching and Control System (VSCS), designed by Harris Corp. of Melbourne, Florida, caused the voice and radio system to shut down, leaving pilots on their own. This situation could have been deadly, as 800 flights were disrupted across the country and planes came dangerously close to each other. Fortunately, commercial jets were equipped with a collision avoidance system, which would alert one pilot to climb and another to ascend

[6] Source: Escobosa, L. Air traffic controller stands watch in carrier air traffic control center aboard aircraft carrier. Photo. *Wikipedia Commons.* 17 Oct. 2008.

when collisions were predicted. However, if this had happened ten or fifteen years earlier, in the absence of the emergency collision avoidance system, several midair collisions could have occurred![7]

What went wrong?

The Federal Aviation Administration (FAA) issued a statement the next day dismissing the failure as human error. The operating and maintenance procedures required the system to be rebooted every 30 days.

However, it was a bug in the system that required the reboot in the first place! The VSCS unit utilized a timer starting at 2^{32} that counted down in milliseconds. Once the timer reached zero, the system would shut down. Counting down from 2^{32} to zero in milliseconds only takes 50 days, so it seemed reasonable to schedule a reboot every month.

After the LA air traffic failure, the FAA tracked the bug and issued a software patch to fix the problem. The system no longer requires a manual reboot, as it takes care of resetting the timer on its own.[8]

How is this related to arrays?

Unfortunately, more details on what exactly caused the system to fail after the timer reached zero have not been released. However, if one is left to hypothesize, the problem could have risen from array indexing errors, as shown below.

[7] Sources: Geppert, L. "Lost Radio Contact Leaves Pilots On Their Own". IEEE Spectrum. Nov. 2006.
Lewis, J. Loftus, W. *Java Software solutions: Foundation of Program Design.* Addison-Wesley.

[8] Sources: Geppert, L. "Lost Radio Contact Leaves Pilots On Their Own". IEEE Spectrum. Nov. 2006.
Lewis, J. Loftus, W. *Java Software solutions: Foundation of Program Design.* Addison-Wesley.

5.1 Example: Air Traffic Control System

You have been contracted by the FAA to track down the notorious bug in the VSCS software that caused the September 2004 failure. Take a look at the code below:

```
#include <iostream>
#include <time.h>  //used to generate the random state

using namespace std;

void cleanLogbook();
int currentState();
void logActivity(int index, char level);
void runTimer();

char logbook[100];  // logbook, used to record activity for the system

int main() {
   srand(time(NULL));  // initialize random seed
   cleanLogbook();
   runTimer();  // start timer
   return 0;
}

// Initializes logbook, cleaning array of possible "junk"
void cleanLogbook() {
   for (int i = 0; i < 100; i++){
      logbook[i] = '0'; // sets every value to 0
   }
}

// Returns the current state or "density" of the air traffic.
// This is an int between 1 to 100, where 1 is low and 100 is high
int currentState(){
   return rand() % 100 + 1; // generate a random int between 1 and 100
}

void runTimer(){
   int den = currentState();

   // Really "big number", for the purposes of this example, set to 100
   // instead of 232
   int bigNumber = 100;
   int timer = bigNumber;

   // Keeps timer running, need to reboot computer to reset timer before timer
   // reaches 0
   while (true){
      if (den < 5) logActivity(timer-1, 'L');
      else if (den > 45 && den < 55) logActivity(timer-1, 'M');
      else if (den > 95) logActivity(timer-1, 'H');
      den = currentState();
      timer--;
   }
}
```

```
// Records the activity level at the given time (index)
void logActivity(int index, char level) {
    logbook[index] = level;
    cout << "State was " << level << " at time " << index << endl;
}
```

Analyzing the code, you notice:

- There is a char array called *logbook*. This char array is the same size as the "big number" 100. Each index in the char array corresponds to a number counting down from 100.
- The *logbook* is cleared at the start of the system via the function *cleanLogbook*. This makes sure that each value in the *logbook* is reset to '0' before starting the timer.
- *currentState* returns *den,* a randomly generated number between 1 and 100. This number is supposed to represent the current state of the air traffic density at that given moment. 1 represents very light traffic density, and 100 represents very heavy traffic density, where density is the term used to define how heavy traffic is at that moment.
- The if-statements in the while-loop in *runTimer* check for three cases of the air traffic density: low, middle, and high density activity. These are the important cases that the FAA[9] needs record in the *logbook*.
- If *den* passes one of the if-statements, a *logActivity* function will be called. If *den < 5*, *logActivity* will record an "L" on the logbook. Similarly, if *45 < den < 55 logActivity* will be called and record an "M". Finally, if *den > 95, logActivity* will record an "H". All of the *logActivity* functions take the current *timer* as the index.
- The timer is initially set to a "large number" and is decreased by 1 after each loop iteration (for the purposes of this example the "large number" we are using is 100 instead of 2^{32}).
- The while-loop loops forever because the program needs to keep track of timing and *logActivity* at all times when it is running.

You see the comment the original programmer left regarding the computer "reboot," asking for the computer to be re-started before the timer reaches zero. However, for debugging purposes you do not follow those instructions and let the timer run.

[9] For the purposes of this example, we are assuming that the FAA requires checking these three cases. This is only a simulation, as the FAA might not truly require these cases, or endorse this simulated example.

1. Run the code once. What happens?

2. Run the code several times. What happens now? What seems to be the same every time you run the program? What is different?

3. When you run the code several times, what is the largest time (index) logged? What is the smallest time (index) logged? What is the problem with this?

4. What is wrong with the *runTimer* function?

If you look at *int timer*, you will notice that after *timer* reaches zero, *timer* continues decreasing. Furthermore, if *den* also happens to fall into one of the three conditions for *logActivity*, one of the *logActivity* functions might get called for a negative index.

5. You explain this to your manager. He suggests an easy fix: when *timer* reaches zero, exit the while loop. Implement this.

6. Is the problem solved? What does your program do now? What is the problem with this?
(Hint: think about the purpose of the program. Is it supposed to loop forever? Is it supposed to terminate?)

7. Now, instead of exiting the while loop when your *timer* reaches zero, you decide to reset the timer to the "big number" again to keep the while loop going. Implement this. What happens now? Is this good or bad?

8. Realizing the fix is not as easy as you initially thought, now you wonder why the previous programmer asked for a manual reboot. Why did he ask for a reboot?
(Hint: Remember that you want to keep logbook accurate).

9. Further investigation on another part of the code reveals that every time the computer is manually shut down, *logbook* is saved into a file and stored in a database. You can add this short function to your code (this code does nothing but print "Saving logbook to database", but for the purposes of this example, you may assume that it saves the array to a file in the database).

```
void saveLogbook(){
    // saves logbook to file
    cout << "Saving logbook to database" << endl;
}
```

Now that you know this, rewrite the code. What does your code do?

10. Run your program. Your program should now be able to run indefinitely without errors. Keep an eye on the *logbook* to make sure that all the entries are correct.

11. Edit your code so that your timer runs 3 iterations and prints out the *logbook* array. Double check that your *logbook* is what it needs to be.

Great job! It looks like there will be no more LA Air Traffic Control failures due to this code!

6. ADDITIONAL EXAMPLES

Note to instructor: Given the variety of possible examples for array range errors, this section includes additional examples that reinforce the same concepts. Depending on the programming experience of your students, some of these examples might be simpler to go through during a section.

6.1 EXAMPLE: PARSON'S PROBLEMS

You are to write a function that takes an array of `int`s and removes every integer that is a multiple of 7 from the array. The function should modify the given size to reflect the new size of the array.

For example, given the following array of `int`s,

1	3	5	7	8	14	4	6	21	2	1

your function should leave the array like this:

1	3	5	8	4	6	2	1

Instead of writing the function from scratch, you must piece together the correct segments of code below to carry out the task. Note that some of the code segments will not be used, and

they are not in any particular order as shown. (You may also repeat any line of code in multiple parts of your answer if you wish.) Your answer must not produce any out-of-bounds errors, even if the resulting array is correctly formed.

```
void removeSevens(int array[], int &size) {

int result[size];
result[j++] = array[i];
int count = size;
count--;
int i = 1;
int i = 0;
int j = size;
while (i >-1) {
while (i < size && j < size) {
for (int j = i; j <= size; j++) {
for (int i = size; i > 0; i--) {
for (i = j; (j <= i && i < size); i++) {
if (array[i] % 7 != 0) {
if (array[i - 1] % 7 == 0) {
if (array[j] % 7 == 0) {
array[i - 1] = array[i];
array[i] = array[j+1];
j++;
j--;
i++;
int result[count];
size--;
```

6.2 Example: Out of Bounds & Overrun

You are starting an internship as a developer for a tech company in Silicon Valley. You have inherited an unfinished project that a previous intern started working on but never finished. When you run your program, something seems to go wrong. You track down a bug in this method:

```
void foo(){
    int i;
    int n[5];

    for (i = 0; i <= 5; i++){
        n[i] = 1;
        printf("n[%d] is %d \n", i, n[i]);
    }
}
```

1. Before you run this code, is there something clearly wrong with this example?

2. Run the code. What happens?

3. What happens if you switch the order of `i` and `n[5]`? Does the result of the program execution change? Why would this be an issue?