GAME2001 Data Structures and Algorithms

Fall 2020

Week 2
Arrays
Basic Searches

Arrays

- most common data storage structure
- built into many programming languages

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

int main(int args, char **argc)
{
    cout << "1D Array Example" << endl << endl;

    const int size = 5;

    // static array implementation
    int array[size] = { 10, 32, 53, 91, 21 };
    cout << " Static array contents (" << size << "): ";
    for(int i = 0; i < size; i++)
    {
        cout << array[i] << " ";
    }
    cout << endl;

    // dynamic array implementation
    int *array2 = new int[size];
    array2[0] = 99;
    array2[1] = 67;
    array2[2] = 23;
    array2[3] = 49;
    array2[4] = 12;
    cout << "Dynamic array contents (" << size << "): ";
    for(int i = 0; i < size; i++)
    {
        cout << array2[i] << " ";
    }
    delete[] array2;

    cout << endl << endl;
    return 1;
}
```

```
1D Array Example

 Static array contents (5): 10 32 53 91 21
Dynamic array contents (5): 99 67 23 49 12

Press any key to continue . . .
```

Issues with Arrays

- Can be static or dynamic
  - With dynamic arrays we must manually create and destroy the array
    - Leads to memory leaks
      - Not deallocating the array
    - Can have dangling pointers
      - Accessing memory that has been deleted through an invalid pointer
- C++ does not perform any bounds checks
  - Can cause unexpected behavior if an attempt is made to access memory outside of the array's range

Custom Unordered Array

```
1  template<class T>
2  class UnorderedArray
3  {
4      public:
5          UnorderedArray(int size, int growBy = 1) :
6              m_array(NULL), m_maxSize(0),
7              m_growSize(0), m_ numElements (0)
8          {
9              if(size)
10             {
11                 m_maxSize = size;
12                 m_array = new T[m_maxSize];
13                 m_growSize = ((growBy > 0) ? growBy : 0);
14             }
15         }
16
17         ~UnorderedArray()
18         {
19             if(m_array != NULL)
20             {
21                 delete[] m_array;
22                 m_array = NULL;
23             }
24         }
25
26     private:
27         T *m_array;
28         int m_maxSize;
29         int m_growSize;
30         int m_numElements;
31 };
```

- Template
- Dynamically allocated
  - Constructor
  - Destructor
- Can grow

```
int GetSize()     { return m_numElements; }
int GetMaxSize()  { return m_maxSize; }
int GetGrowSize() { return m_growSize; }

void SetGrowSize(int val)
{
    assert(val >= 0);
    m_growSize = val;
}
```

Custom Unordered Array

- Inserting
  - At the end of the array

```
1 template<class T>
2 class UnorderedArray
3 {
4     public:
5         virtual void push(T val)
6         {
7             assert(m_array != NULL);
8
9             if(m_numElements >= m_maxSize)
10            {
11                Expand();
12            }
13
14            m_array[m_numElements] = val;
15            m_numElements++;
16        }
17 };
```

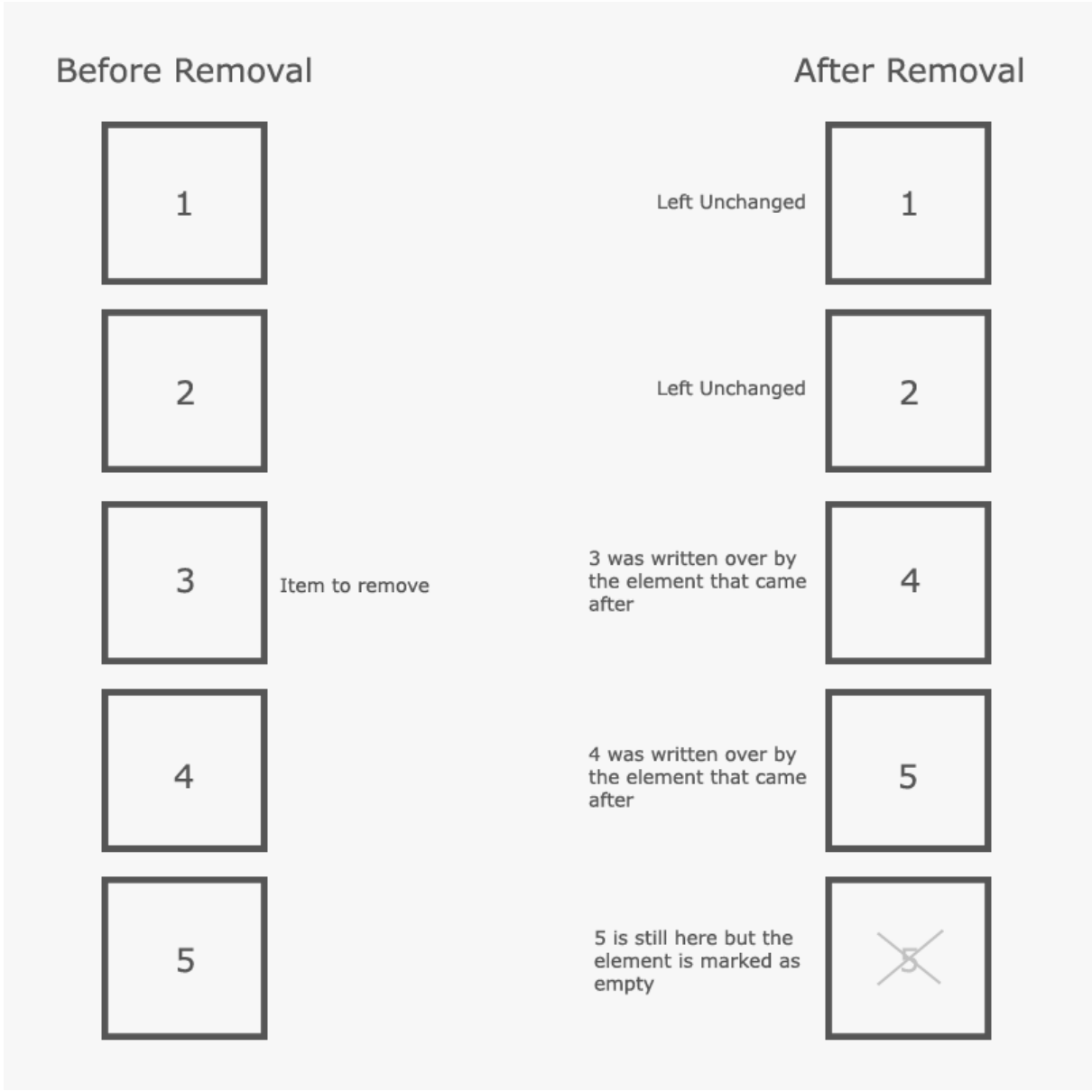
- What is Big-O notation?

Custom Unordered Array

- Deletion 3 options
  - Option1 (conserves memory, huge performance penalty)
    - Resize itself and only stores the data that was in the original array minus the item to be removed
      - create a new array
      - copy over the data from the original array up to the point of the object being deleted
      - copy the data after the object being deleted one index down
      - delete the old memory

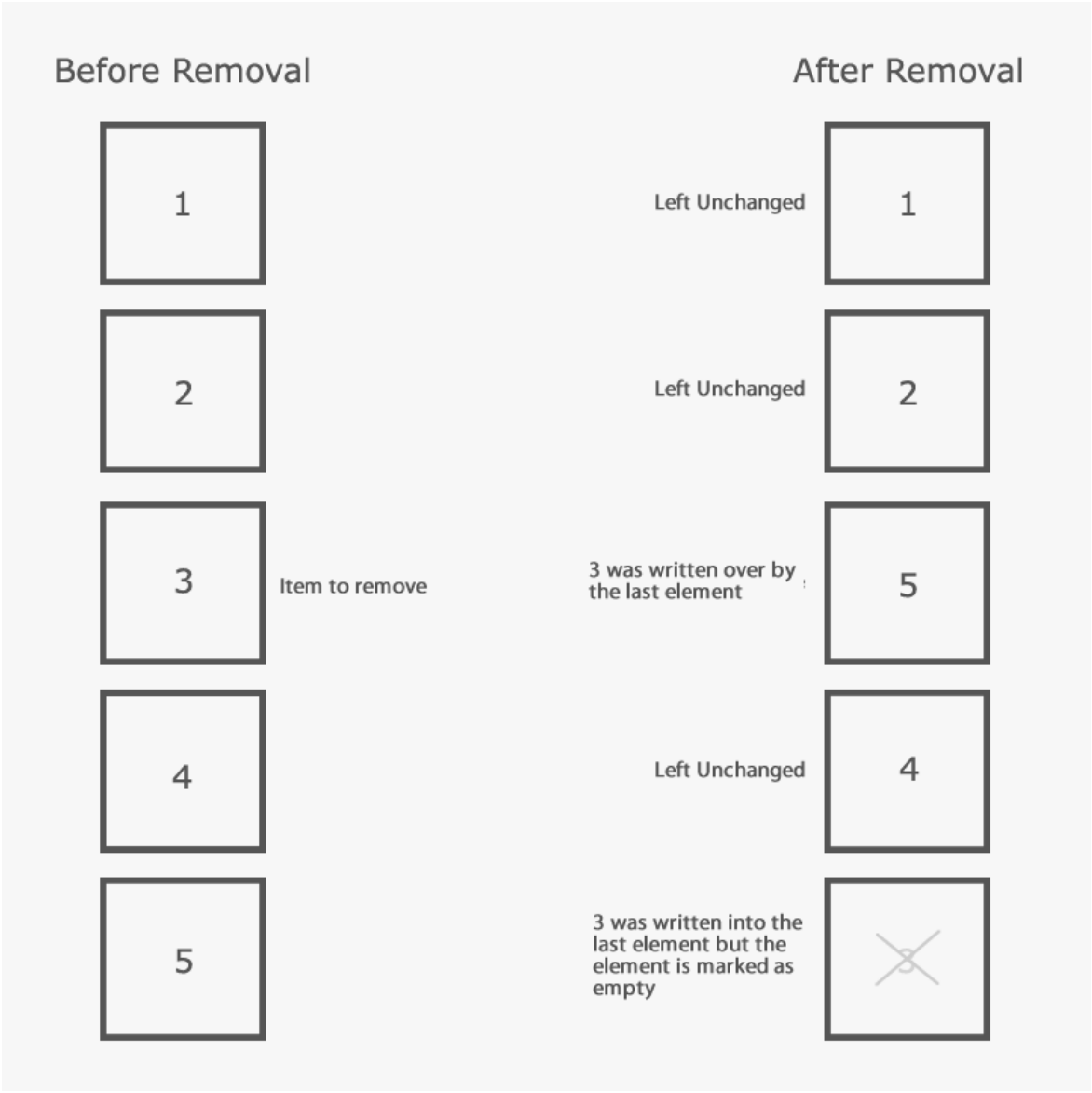
Custom Unordered Array

- Option 2 (no memory allocation and deletion)
  - copy the data that comes after the item to be deleted one element down
  - item to be removed is overridden with the indexes that come after it
  - leaving an empty spot on the top of the array
  - still need to perform a copy operation on the array


Before Removal
After Removal
1
2
3
Item to remove
4
5
Left Unchanged
1
Left Unchanged
2
3 was written over by the element that came after
4
4 was written over by the element that came after
5
5 is still here but the element is marked as empty

Custom Unordered Array

- Option 3
  - swap the last element with the one being deleted
  - only for unordered arrays that you don't care about their place in the array


Before Removal
After Removal
1
Left Unchanged
1
2
Left Unchanged
2
3
Item to remove
3 was written over by the last element
5
4
Left Unchanged
4
5
3 was written into the last element but the element is marked as empty

```
template<class T>
class UnorderedArray
{
    public:
        // removes the last item that was inserted
        void pop()
        {
            if(m_numElements > 0)
                m_numElements--;
        }

        // removes an index from the array by overriding the index to delete
        void remove(int index)
        {
            assert(m_array != NULL);

            if(index >= m_maxSize)
                return;

            for(int k = index; k < m_maxSize - 1; k++)
                m_array[k] = m_array[k + 1];

            m_maxSize--;

            if(m_numElements >= m_maxSize)
                m_numElements = m_maxSize - 1;
        }

        // removes all items from the array
        void clear()
        {
            m_numElements = 0;
        }
};
```

Custom Unordered Array

- Expanding
  - Expanding the array gives the container a dynamically growing heap of memory that adjusts as the need arises
    - create a bigger new array
    - copy the data from the original memory location to the new one (performance hit)
    - delete the old memory
  - Can expand
    - By 1
    - By a grow by value (5)
    - By increasing values (2, 4, 8, 16, 32)

Custom Unordered Array

```
1  template<class T>
2  class UnorderedArray
3  {
4      private:
5          bool Expand()
6          {
7              if(m_growSize <= 0)
8                  return false;
9
10             T *temp = new T[m_maxSize + m_growSize];
11             assert(temp != NULL);
12
13             memcpy(temp, m_array, sizeof(T) * m_maxSize);
14
15             delete[] m_array;
16             m_array = temp;
17
18             m_maxSize += m_growSize;
19
20             return true;
21         }
22 };
```

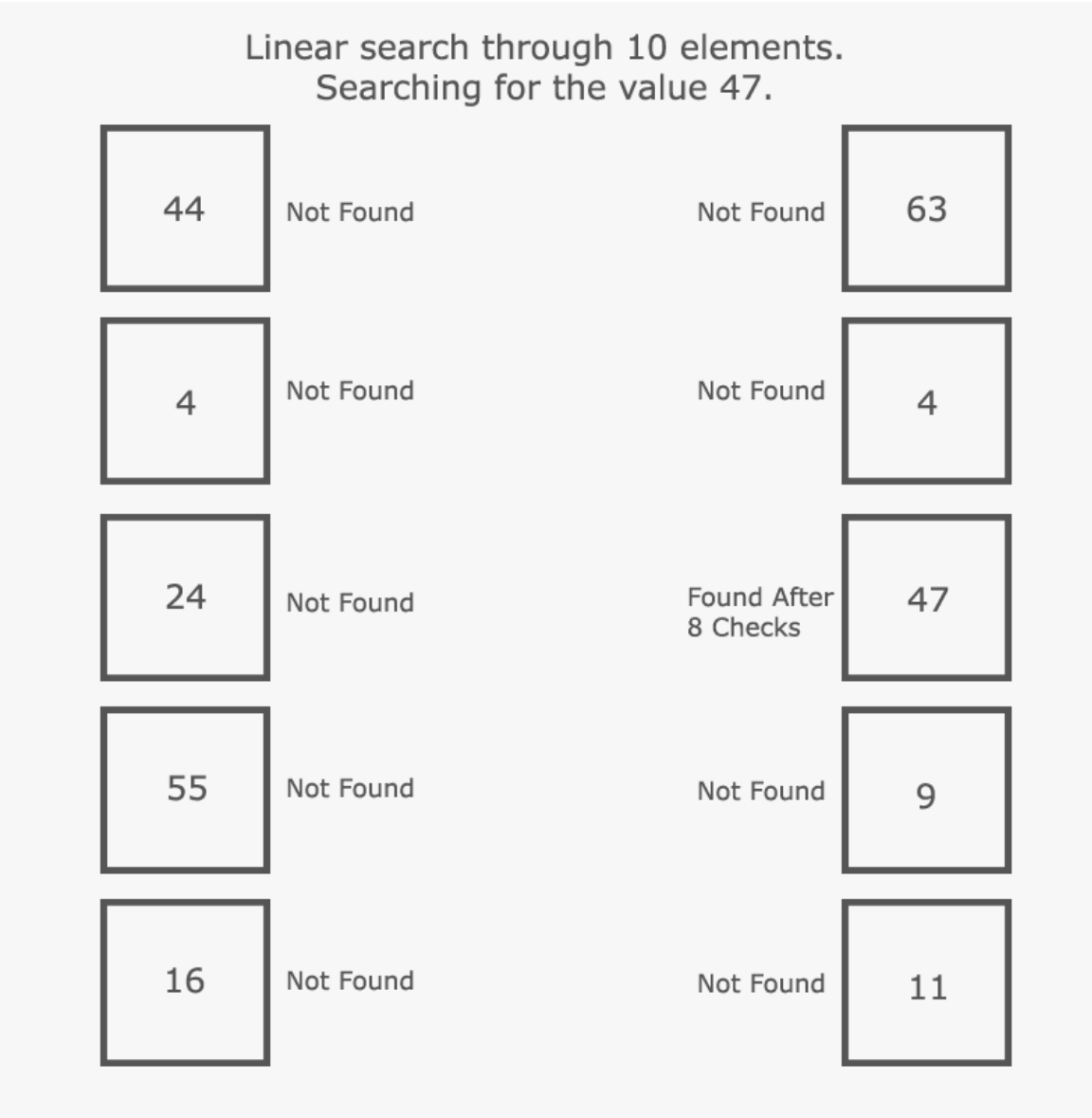
Custom Unordered Array

- Array Style Access
  - out-of-bound errors (debug only)
    - causes performance issues in release
  - Returns a reference (mimics traditional arrays)

```
1  template<class T>
2  class UnorderedArray
3  {
4      public:
5          T& operator[](int index)
6          {
7              assert(m_array != NULL && index < m_numElements);
8              return m_array[index];
9          }
10 };
```

Custom Unordered Array

- Basic search – Linear search
  - Main option for unordered arrays
  - Brute-force-style search
  - stepping through each element of the array
    - checking to see if the value of that element matches the value of what is being searched for
      - If found return the index
      - If not go to the next element until you reach the end of the array


Linear search through 10 elements.
Searching for the value 47.
44
Not Found
Not Found
63
4
Not Found
Not Found
4
24
Not Found
Found After
8 Checks
47
55
Not Found
Not Found
9
16
Not Found
Not Found
11

Custom Unordered Array

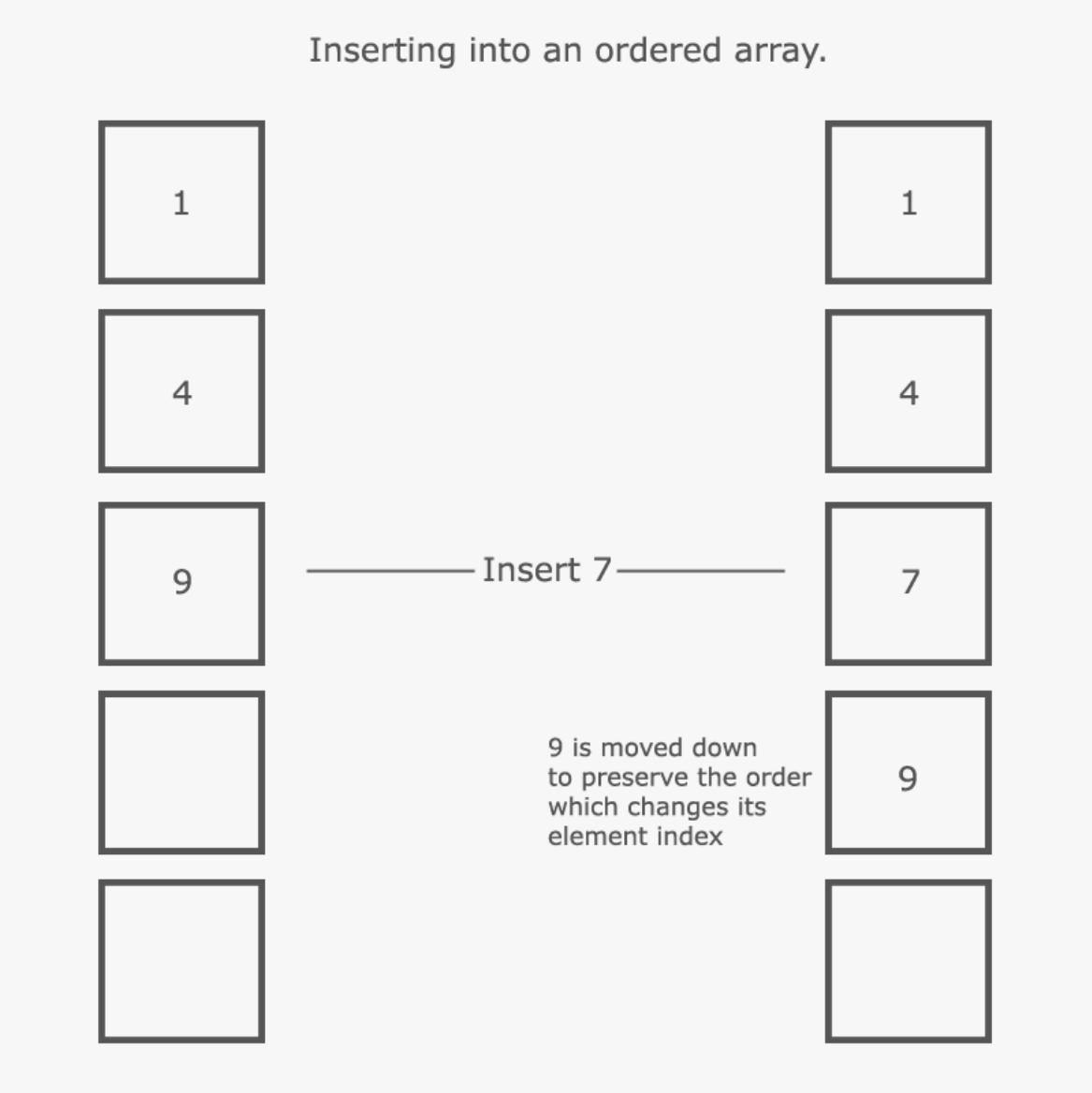
- Linear Search

```
1  template<class T>
2  class UnorderedArray
3  {
4      int search(T val)
5      {
6          assert(m_array != NULL);
7
8          for(int i = 0; i < m_numElements; i++)
9          {
10             if(m_array[i] == val)
11                 return i;
12         }
13
14         return -1;
15     }
16 };
```

```
1  #include<iostream>
2  #include"Arrays.h"
3
4  using namespace std;
5
6  int main(int args, char **argc)
7  {
8      UnorderedArray<int> array(3);
9
10     array.push(3);
11     array.push(53);
12     array.push(83);
13     array.push(23);
14     array.push(82);
15
16     array[2] = 112;
17
18     array.pop();
19     array.remove(2);
20
21     cout << "Unordered array contents: ";
22
23     for(int i = 0; i < array.GetSize(); i++)
24     {
25         cout << array[i] << " ";
26     }
27
28     cout << endl;
29
30     cout << "Search for 53 was found at index: ";
31     cout << array.search(53);
32
33     cout << endl << endl;
34
35    return 1;
36 }
```

Custom Ordered Arrays

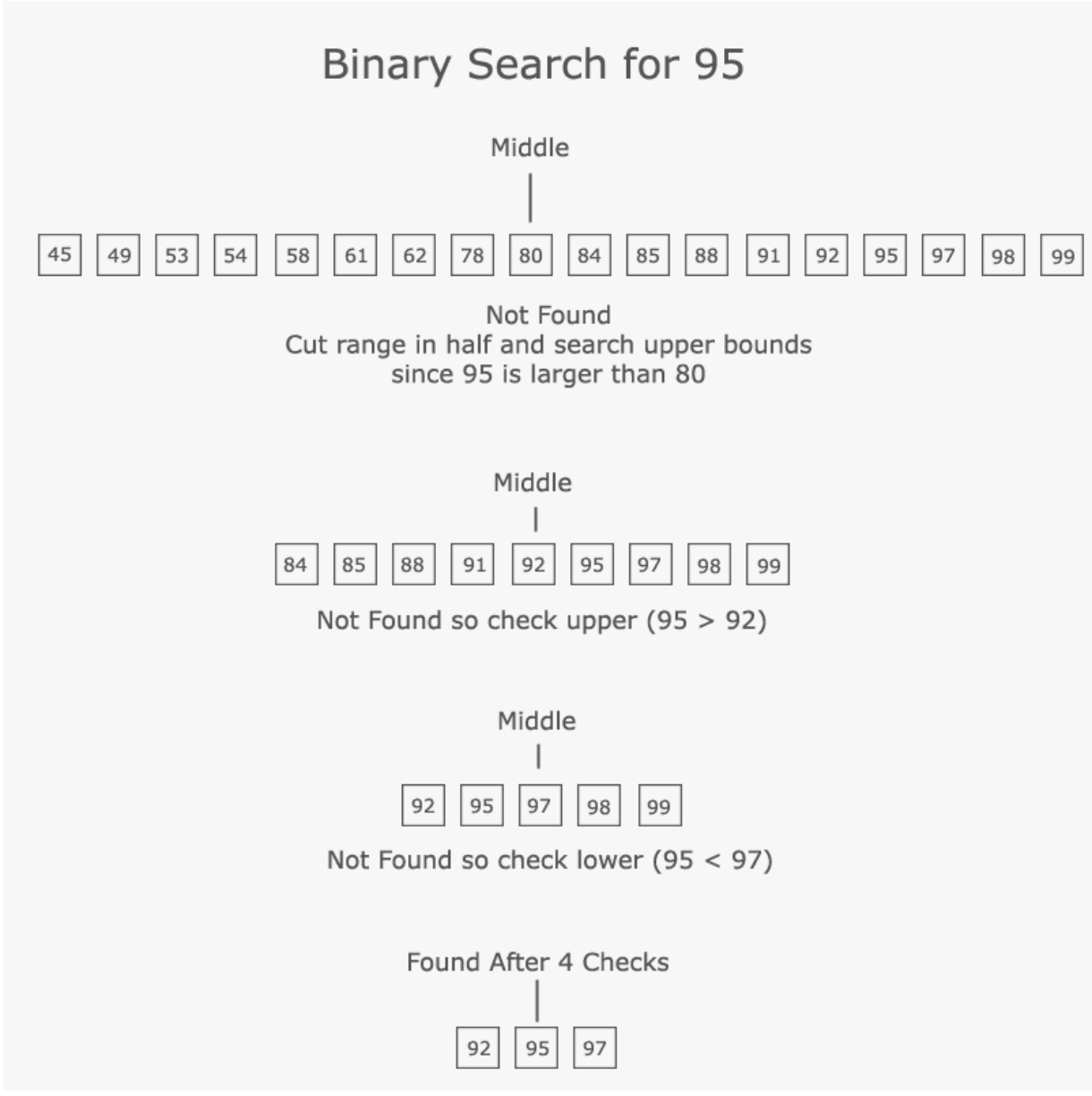
- an array that has its contents in some kind of order
  - larger to smaller or vice versa
  - determined during the items' insertion


Inserting into an ordered array.
1
4
9
Insert 7
7
9 is moved down
to preserve the order
which changes its
element index
1
4
9

```
1  template <class T>
2  class OrderedArray
3  {
4      public:
5          int push(T val)
6          {
7              assert(m_array != NULL);
8
9              if(m_numElements >= m_maxSize)
10             {
11                 Expand();
12             }
13
14             int i, k;
15
16             for(i = 0; i < m_numElements; i++)
17             {
18                 if(m_array[i] > val)
19                     break;
20             }
21
22             for(k = m_numElements; k > i; k--)
23             {
24                 m_array[k] = m_array[k - 1];
25             }
26
27             m_array[i] = val;
28             m_numElements++;
29
30             return i;
31         }
32 };
```

Custom Ordered Arrays

- Basic search – Binary search
  - performed on an ordered array
  - taking a value that is being searched for and testing it by the element in the middle of the array
    - If lower, it might exist in the first half of the array
    - If higher, it might exist in the upper half of the array
  - once the direction is known, half of that new section is tested to further narrow down where the value can be
  - repeated until the value is found or until there are no more elements left


Binary Search for 95
Middle
45 49 53 54 58 61 62 78 80 84 85 88 91 92 95 97 98 99
Not Found
Cut range in half and search upper bounds
since 95 is larger than 80
Middle
84 85 88 91 92 95 97 98 99
Not Found so check upper (95 > 92)
Middle
92 95 97 98 99
Not Found so check lower (95 < 97)
Found After 4 Checks
92 95 97

```
1  template <class T>
2  class OrderedArray
3  {
4      public:
5          int search(T searchKey)
6          {
7              assert(m_array != NULL);
8
9              int lowerBound = 0;
10             int upperBound = m_numElements - 1;
11             int current = 0;
12
13             while(1)
14             {
15                 current = (lowerBound + upperBound) >> 1;
16
17                 if(m_array[current] == searchKey)
18                 {
19                     return current;
20                 }
21                 else if(lowerBound > upperBound)
22                 {
23                     return -1;
24                 }
25                 else
26                 {
27                     if(m_array[current] < searchKey)
28                         lowerBound = current + 1;
29                     else
30                         upperBound = current - 1;
31                 }
32             }
33
34             return -1;
35         }
36 };
```

Custom Ordered Array

- Array Style Access
  - out-of-bound errors (debug only)
    - causes performance issues in release
  - Returns a const reference (mimics traditional arrays)
    - does not allow changing the value in the array
      - can unintentionally unorder the array

```
1  template <class T>
2  class OrderedArray
3  {
4      public:
5          // Making this const allows for reading but not writing.
6          const T& operator[](int index)
7          {
8              assert(m_array != NULL && index < m_numElements);
9              return m_array[index];
10         }
11 };
```

```
1  #include<iostream>
2  #include"Arrays.h"
3
4  using namespace std;
5
6  int main(int args, char **argc)
7  {
8      OrderedArray<int> array(3);
9
10     array.push(43);
11     array.push(8);
12     array.push(23);
13     array.push(94);
14     array.push(17);
15
16     array.pop();
17     array.remove(2);
18
19     cout << "Ordered array contents: ";
20
21     for(int i = 0; i < array.GetSize(); i++)
22     {
23         cout << array[i] << " ";
24     }
25
26     cout << endl;
27
28     cout << "Search for 12 was found at index: ";
29     cout << array.search(12);
30
31     cout << endl << endl;
32
33    return 1;
34 }
```

Dealing with duplicates

- C++ arrays cannot check for duplicate data
- one option is to search the list before inserting new items into it
  - If an item exists, the insertion can be avoided

```
1 if(array.Search(3) != -1)
2     array.push(3);
3
4 if(array.Search(53) != -1)
5     array.push(53);
6
7 if(array.Search(83) != -1)
8     array.push(83);
```

Pros & Cons

- Ordered Arrays
  - Faster to search (binary search)
- Unordered Arrays
  - Faster Insertion time

- When insertions are rare but searches are frequent use ordered arrays

STL Arrays

Method and Operator Names	Descriptions
`vector<type>()`	Constructor that creates an empty vector container
`vector<type>(n)`	Constructor that creates a vector of size *n*
`vector<type>(source)`	Copy constructor that creates a vector that is a copy of the `source` object
`vector<type>(n, val)`	Constructor that creates a vector of size *n* that has its elements initialized to `val`
`vector<type> (src.begin, src.end)`	Constructor that creates a vector out of the elements in the `src` vector defined by its beginning and ending iterators
`~vector<type>()`	Destructor that destroys the vector
`assign(n, val)`	Assigns to the vector *n* elements of the value `val`
`at(index)`	Returns a reference to the element specified by `index`
`back()`	Returns a reference to the last element
`begin()`	Returns a random-access iterator to the beginning of the vector
`capacity()`	Returns the number of elements the vector could contain without needing to allocate more memory
`clear()`	Erases the elements of a vector
`empty()`	Returns true if the vector is empty, or else false
`end()`	Returns a random-access iterator to the end of the vector

STL Arrays

`erase(index)`	
`erase(begin, end)`	Erases the element specified by `index` or a range of elements specified by the iterators `begin` and `end`
`front()`	Returns a reference to the beginning of the first element in the vector
`get_allocator()`	Returns the allocator used by the vector container
`insert(index, val)`	
`insert(index, n, val)`	
`insert(index, begin, end)`	Inserts a value specified by `val` or a range of values specified by the `begin` and `end` iterators into the position `index`. `N` is the total number of times to insert into the container.
`max_size()`	Returns the maximum size of the container
`push_back(val)`	Adds a value `val` to the end of the container
`pop_back()`	Removes the value at the end of the container
`rbegin()`	Returns an iterator to the first element in a reverse vector container
`rend()`	Returns an iterator to the last element in a reverse vector container
`resize(n)`	Specifies a new size n for the container
`reserve(n)`	Reserves the minimum size n for the container
`size()`	Returns the number of elements in the container

STL Arrays

`swap(source)`	Swaps two vectors with one another
`operator[index]`	Returns a reference to an element at `index`
`operator==`	Boolean operator that returns `true` if two vectors are equal, or else it returns `false`
`operator!=`	Boolean operator that returns `true` if two vectors are not equal, or else it returns `false`
`operator<`	Boolean operator that returns `true` if the first vector is less than the second
`operator>`	Boolean operator that returns `true` if the first vector is greater than the second
`operator<=`	Boolean operator that returns `true` if the first vector is less than or equal to the second
`operator>=`	Boolean operator that returns `true` if the first vector is greater than or equal to the second

STL Algorithms

`accumulate(begin, end, val)`	Returns the sum of all of the elements in the range of the `begin` and `end` iterators (and adds `val` to each element)
`copy(src.begin, src.end, dst.begin)`	Copies the elements in `src` to `dst` in the range of the `src`'s `begin` and `end` iterators
`copy_backward(src.begin, src.end, dst.begin)`	Same as `copy()` but with the elements in reverse
`count(begin, end, val)`	Counts the number of elements that match `val` in the range specified by the `begin` and `end` iterators
`fill(begin, end, val)`	Fills a container in the range specified by `begin` and `end` with the value `val`
`find(begin, end, val)`	Returns an input iterator to the first occurrence of `val` in the range specified by `begin` and `end`
`min_element(begin, end)`	Returns the minimum element in the range specified by `begin` and `end`
`max_element(begin, end)`	Returns the maximum element in the range specified by `begin` and `end`

STL Algorithms

`random_shuffle(begin, end)`	Randomly shuffles elements in a range
`remove(begin, end, val)`	Removes a value `val` from the container in a range without changing the order of the remaining elements
`replace(begin, end, oldVal, newVal)`	Replaces the elements that match `oldVal` with `newVal` in a range
`reverse(begin, end)`	Changes the order of the elements in the specified range
`search (begin1, end1, begin2, end2)`	Searches for the first occurrence of a set of values specified in `begin2` and `end2` with those in `begin1` and `end1`
`sort (begin, end)`	Sorts the elements in the range into ascending order
`swap(vec1, vec2)`	Swaps the elements between two vectors
`unique(begin, end)`	Removes duplicate elements within the range

STL Vector Example

```cpp
#include<iostream>
#include<vector>

using namespace std;

void PrintVector(vector<int> &array)
{
    cout << "Contents (" << "Size: " << (int)array.size() <<
           " Max: " << (int)array.capacity() << ") - ";

    for(int i = 0; i < (int)array.size(); i++)
    {
        cout << array[i] << " ";
    }

    cout << endl;
}
```

```
STL Vector Example

  Inserted into vector.  Contents (Size: 4 Max: 5) - 10 20 30 40
Popped two from vector.  Contents (Size: 2 Max: 5) - 10 20
        Cleared vector.  Contents (Size: 0 Max: 5) -

Vector is empty.
```

```cpp
int main(int args, char **argc)
{
    cout << "STL Vector" << endl << endl;

    vector<int> array;
    array.reserve(5);

    array.push_back(10);
    array.push_back(20);
    array.push_back(30);
    array.push_back(40);

    cout << "  Inserted into vector.  ";
    PrintVector(array);

    array.pop_back();
    array.pop_back();

    cout << "Popped two from vector.  ";
    PrintVector(array);

    array.clear();

    cout << "        Cleared vector.  ";
    PrintVector(array);

    cout << endl;

    if(array.empty() == true)
       cout << "Vector is empty" << endl;
    else
       cout << "Vector is NOT empty" << endl;

    return 1;
}
```

STL Iterators

- similar to pointers in that they point to elements in a container
- very important when using the STL
- work on sequence containers, which include our vector class
- include container iterators (forward, bidirectional, and random access), input iterators, and output iterators

STL Iterator Example

```
#include<iostream>
#include<vector>
#include<algorithm>
#include<numeric>    //accumulate()

using namespace std;

void PrintVector(vector<int> &array)
{
    cout << "Contents (" << "Size: " << (int)array.size()
         <<  " Max: " << (int)array.capacity() << ") - ";

    ostream_iterator<int> output(cout, " ");
    copy(array.begin(), array.end(), output);

    cout << endl;
}

int main(int args, char **argc)
{
   cout << "STL Iterators" << endl;

   vector<int> array;
   array.reserve(5);

   array.push_back(10);
   array.push_back(20);
   array.push_back(30);
   array.push_back(40);
   array.push_back(50);

   // Calling the copy algorithm.
   vector<int> array2;
   for(int i = 0; i < 5; i++)
      array2.push_back(0);

   copy(array.begin(), array.end(), array2.begin());

   cout << "  Inserted into vector:  ";
   PrintVector(array);

   // Run the accumulate algorithm.
   cout << "             Accumulate:  "
        << accumulate(array.begin(), array.end(), 0)
        << endl;

   array.pop_back();
   array.pop_back();

   cout << "Popped two from vector:  ";
   PrintVector(array);

   // Clear the container.
   array.clear();

   cout << "        Cleared vector:  ";
   PrintVector(array);
   cout << endl;

   // Test if the container is empty.
   if(array.empty() == true)
      cout << "Vector is empty" << endl;
   else
      cout << "Vector is NOT empty" << endl;

   return 1;
}
```