Patterns: Working with Arrays

Steven Zeil

October 14, 2013

Contents

1 Static & Dynamic Allocation

1.1 Static Allocation

Static Allocation

```
const int MaxItems = 100;

// Names of items
std::string itemNames[MaxItems];
```

- Simple
- Best suited to situations where we know the number of items at compile time

```
string monthNames[12];
```

. . . . . .

How Many Elements Do We Need?

If we don't know how many elements we really need, we need to guess (??)

- Too few, and the program crashes
- Too many, and we waste space (and maybe time)

. . . . . .

1.2 Dynamic Allocation

Dynamic Allocation

- Use pointer to array

- Allocated on heap
- Needs to be deleted afterwards

.....

Example: bidders.h

```
extern int nBidders;
extern Bidder* bidders;

void readBidders (std::istream& in);
```

.....

Example: bidders.cpp

```
int nBidders;
Bidder* bidders;


/**
 * Read all bidders from the indicated file
 */
void readBidders (istream& in)
{
    in >> nBidders;                          ❶
    bidders = new Bidder[nBidders];          ❷
    for (int i = 0; i < nBidders; ++i)
    {
     read (in, bidders[i]);                  ❸
    }
}
```


```
void cleanUpBidders()
{
  delete [] bidders;
}
```

❶ Here we read the desired size of the arrays from the input

❷ Here we use that value to actually allocate arrays of the desired size

❸ Once that is done, we can access the arrays in the usual manner using the [ ].

Dynamic Array Declarations are Confusing

```
extern Bidder* bidders;
```

- We can't tell by looking at this declaration whether it is intended to point to
  - a single bidder, or
  - an array of bidders
- Need to rely on documentation

Why Are Arrays Different?

Because arrays are "really" pointers...

- `array1 = array2;` does *not* copy the array
- When arrays are passed to functions, we are actually passing a pointer
  - passing arrays is fast and efficient

– changes to the array can be seen by the caller

```
int x;
int arr[100];
void foo (int i, int* a);
   ⋮
foo(x, a);
   ⋮
void foo (int i, int* a)
{
  i = i + 1; // does not affect x
  a[0] = i;  // does affect arr
}
```

.....

2 Partially Filled Arrays

In our prior examples, we have typically dealt with arrays that were just large enough to hold the data we wanted, or that were filled immediately to a certain fixed size and then, so long as we continued working with the array, that size never varied.

Now we turn to another common pattern: sometimes we add and remove elements to arrays as we progress with our algorithm. We might not even know ahead of time how many elements we will wind up with until we reach the end of the program.

Consider, for example, a spell checker that needs to accumulate a list of misspelled words from a document. That list would start out at zero. As we discover words in the document that are not in our dictionary, we would add them, one by one, to our list of misspellings. We won't be able to predict just how long the list will get until we've finished the spellcheck.

Partially Filled Arrays

To work with a *partially filled* array, we generally need to have access to

- the array itself
- an integer counter indicating how many elements are currently in the array

  - the *size* or *length* of the list

- an integer counter indicating the maximum nmber of elements we can have without overflowing the array
  - the *capacity* of the list

. . . . . .

With that in mind, let's look at some typical operations on partially filled arrays.

2.1 Adding Elements

Adding Elements

Variations include

- adding to the end
- adding in the middle
- adding in order

. . . . . .

Add to the end

```
void addToEnd (std::string* array, int& size, std::string value)
{
   array[size] = value;
   ++size;
}
```

- Assumes that we have a separate integer indicating how many elements are in the array
- and that the "true" size of the array is at least one larger than the current value of that counter

. . . . . .

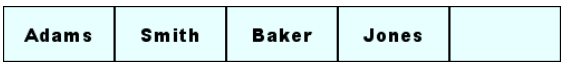
Add to the middle

- addElement (array, size, index, value)
- Adds `value` into `array[index]`, shifting all elements already in positions `index..size-1` up one, to make room.
- Increments the size variable

If we have this and we do

```
addElement (array, 3, 1, "Smith");
```

we should get this.



Add to the middle: implementation

```
// Add value into array[index], shifting all elements already in positions
//    index..size-1 up one, to make room.
//  - Assumes that we have a separate integer (size) indicating how
//     many elements are in the array
//  - and that the "true" size of the array is at least one larger
//      than the current value of that counter

void addElement (std::string* array, int& size, int index, std::string value)
{
  // Make room for the insertion
  int toBeMoved = size - 1;
  while (toBeMoved >= index) {
    array[toBeMoved+1] = array[toBeMoved];
    --toBeMoved;
  }
  // Insert the new value
```

```
  array[index] = value;
  ++size;
}
```

- You can try out the `addToEnd` and `addElement` functions here.
  - Try them out with different inputs until you understand how they work.

.....

Add in order

- `int addInOrder (array, size, value)`
- Assume the elements of the array are already in order
- Find the position where `value` could be added to keep everything in order, and insert it there.
- Return the position where it was inserted

If we have this and we do

```
addInOrder (array, 3, "Clarke");
```

we should get this

.....

Add in order implementation

This works:

```
int addInOrder (std::string* array, int& size,
    std::string value)
{
  // Find where to insert
```


```
  int pos = 0;
  while (pos < size && value > array[pos])
    ++pos;
  addElement (array, size, pos, value);
  return pos;
}
```

.....

Add in order (streamlined)

This is a bit faster:

```
int addInOrder ((std::string* array, int& size,
    std::string value)
{
  // Make room for the insertion
  int toBeMoved = size - 1;
  while (toBeMoved >= 0 && value < array[toBeMoved]) {
    array[toBeMoved+1] = array[toBeMoved];
    --toBeMoved;
  }
  // Insert the new value
  array[toBeMoved+1] = value;
  ++size;
  return toBeMoved+1;
}
```

Try This: You can try out the two addInOrder functions here.

.....

2.2 Searching for Elements

Sequential Search

Search an array for a given value, returning the index where found or -1 if not found.


```
int seqSearch(const int list[], int listLength, int searchItem)
{
   int loc;
   bool found = false;

   for (loc = 0; loc < listLength; loc++)
      if (list[loc] == searchItem)
        {
         found = true;
         break;
        }

   if (found)
      return loc;
   else
      return -1;
}
```

- How many elements does this visit in the worst case?
- How many elements does this visit on average?
  - if `searchItem` is in the array?
  - if `searchItem` is not in the array?

.....

Sequential Search 2

Search an array for a given value, returning the index where found or -1 if not found.

```
int seqSearch(const int list[], int listLength, int searchItem)
{
```


```
    int loc;

    for (loc = 0; loc < listLength; loc++)
        if (list[loc] == searchItem)
            return loc;

    return -1;
}
```

.....

Sequential Search (Ordered Data)

Search an array for a given value, returning the index where found or -1 if not found.

- If data is in order, we can stop early
  - as soon as `list[loc] > searchItem`

.....

seqOrderedSearch

```
int seqOrderedSearch(const int list[], int listLength, int searchItem)
{
    int loc = 0;

    while (loc < listLength && list[loc] < searchItem)
      {
       ++loc;
      }
    if (loc < listLength && list[loc] == searchItem)
       return loc;
    else
       return -1;
}
```


- How many elements does this visit in the worst case?
- How many elements does this visit on average?
  - if `searchItem` is in the array?
  - if `searchItem` is not in the array?

Try This: You can try out the sequential search and sequential ordered search functions here.

.....

2.3 Removing Elements

Removing Elements

- `removeElement (std::string* array, int& size; int index)`
- ```
  void removeElement (std::string* array, int& size,
      int index)
  {
    int toBeMoved = index + 1;
    while (toBeMoved < size) {
      array[toBeMoved-1] = array[toBeMoved];
      ++toBeMoved;
    }
    --size;
  }
  ```

Try This: You can try out the removeElement function here.

.....


  ```

3 Arrays and Templates

arrayUtils - first draft

The functions we have developed in this section can be used in many different programs, so it might be useful to collect them into an "Array Utilities" module.

- Header:

```
#ifndef ARRAYUTILS0_H
#define ARRAYUTILS0_H

#include <string>

//  Add to the end
//  - Assumes that we have a separate integer (size) indicating how
//     many elements are in the array
//  - and that the "true" size of the array is at least one larger
//      than the current value of that counter
void addToEnd (std::string* array, int& size, std::string value);



// Add value into array[index], shifting all elements already in positions
//    index..size-1 up one, to make room.
//  - Assumes that we have a separate integer (size) indicating how
//     many elements are in the array
//  - and that the "true" size of the array is at least one larger
//      than the current value of that counter

void addElement (std::string* array, int& size, int index, std::string value);
```

```
// Assume the elements of the array are already in order
// Find the position where value could be added to keep
//    everything in order, and insert it there.
// Return the position where it was inserted
//  - Assumes that we have a separate integer (size) indicating how
//     many elements are in the array
//  - and that the "true" size of the array is at least one larger
//      than the current value of that counter

int addInOrder (std::string* array, int& size, std::string value);



// Search an array for a given value, returning the index where
//    found or -1 if not found.
int seqSearch(const int list[], int listLength, int searchItem);



// Search an ordered array for a given value, returning the index where
//    found or -1 if not found.
int seqOrderedSearch(const int list[], int listLength, int searchItem);


// Removes an element from the indicated position in the array, moving
// all elements in higher positions down one to fill in the gap.
void removeElement (std::string* array, int& size, int index);


#endif
```


- Compilation unit:

```
#include "arrayUtils0.h"



//  Add to the end
//  - Assumes that we have a separate integer (size) indicating how
//     many elements are in the array
//  - and that the "true" size of the array is at least one larger
//      than the current value of that counter
void addToEnd (std::string* array, int& size, std::string value)
{
   array[size] = value;
   ++size;
}



// Add value into array[index], shifting all elements already in positions
//    index..size-1 up one, to make room.
//  - Assumes that we have a separate integer (size) indicating how
//     many elements are in the array
//  - and that the "true" size of the array is at least one larger
//      than the current value of that counter

void addElement (std::string* array, int& size, int index, std::string value)
{
  // Make room for the insertion
  int toBeMoved = size - 1;
  while (toBeMoved >= index) {
```


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


// Assume the elements of the array are already in order
// Find the position where value could be added to keep
//    everything in order, and insert it there.
// Return the position where it was inserted
//  - Assumes that we have a separate integer (size) indicating how
//     many elements are in the array
//  - and that the "true" size of the array is at least one larger
//      than the current value of that counter

int addInOrder (std::string* array, int& size, std::string value)
{
  // Make room for the insertion
  int toBeMoved = size - 1;
  while (toBeMoved >= 0 && value < array[toBeMoved]) {
    array[toBeMoved+1] = array[toBeMoved];
    --toBeMoved;
  }
  // Insert the new value
  array[toBeMoved+1] = value;
  ++size;
  return toBeMoved+1;
}
```


```
// Search an array for a given value, returning the index where
//    found or -1 if not found.
int seqSearch(const int list[], int listLength, int searchItem)
{
    int loc;

    for (loc = 0; loc < listLength; loc++)
        if (list[loc] == searchItem)
            return loc;

    return -1;
}


// Search an ordered array for a given value, returning the index where
//    found or -1 if not found.
int seqOrderedSearch(const int list[], int listLength, int searchItem)
{
    int loc = 0;

    while (loc < listLength && list[loc] < searchItem)
      {
       ++loc;
      }
    if (loc < listLength && list[loc] == searchItem)
       return loc;
    else
       return -1;
}
```


```
// Removes an element from the indicated position in the array, moving
// all elements in higher positions down one to fill in the gap.
void removeElement (std::string* array, int& size, int index)
{
  int toBeMoved = index + 1;
  while (toBeMoved < size) {
    array[toBeMoved] = array[toBeMoved+1];
    ++toBeMoved;
  }
  --size;
}
```

- Test driver:

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

#include "arrayUtils.h"

using namespace std;

void doTest (char** data, int size)
{
  string array[size];
  int aSize = 0;
  for (int i = 0; i < size; ++i)
    {
      addInOrder (array, aSize, string(data[i]));
    }
```


```
  for (int i = 0; i < size; ++i)
    {
      int index = seqOrderedSearch (array, aSize, string(data[i]));
      cout << data[i] << " was inserted at position "
       << index << endl;
    }

}


int main (int argc, char** argv)
{
  doTest (argv+1, argc-1);
  return 0;
}
```

Question: The main function will not compile. Why?

.....

Answer: We declared `seqOrderedSearch` to work on arrays of *int*, but the main program tries to use it on an array of strings.

. . . . . .

Overloading

One solution is to provide different versions of each function for each kind of array:

```
// Search an ordered array for a given value, returning the index where
//    found or -1 if not found.
int seqOrderedSearch(const int list[], int listLength, int searchItem);
int seqOrderedSearch(const char list[], int listLength, char searchItem);
int seqOrderedSearch(const double list[], int listLength, double searchItem);
int seqOrderedSearch(const float list[], int listLength, float searchItem);
int seqOrderedSearch(const std::string list[], int listLength, std::string searchItem);
```

with nearly identical function bodies for each one.

- called *overloading* the function name
  - Tedious
  - Not possible to cover all possible data types

. . . . . .

Function Templates

A *function template* is a *pattern* for an infinite number of possible functions.

- Contains special symbols called *template parameters* that function like blank spaces in a form
- Symbols can be replaced to "fill out" the form
  - Called *instantiating* the template

. . . . . .

A Simple Template

```
template <typename T>
void swap (T & x, T & y)
{
  T temp = x;
  x = y;
  y = temp;
}
```

- The template header announces that is a template.
  - Without this, it would look like an ordinary function
- The header lists the template parameters for this pattern.
  - These will be replaced when the template is used (instantiated).
  - Can have more than one (comma-separated)
  - Each preceded by the word `typename` or `class`
- Each template parameter *must* appear as a type somewhere in the function's parameter list
- The `swap` template is declared in the `std` library header `<algorithm>`

.....

Using A Function Template

- Import it from the appropriate header (if not declared locally)
- Call it with the desired parameters
  - Compiler figures out what substitutions to make

```
#include <algorithm>
using namespace std;
   ⋮
string a = "abc";
string b = "bcde";
swap (a, b); // compiler replaces "T" by "string"
int i = 0;
int j = 2;
swap (i, j); // compiler replaces "T" by "int"
```

.....

Some Other std Function Templates

- `min(x,y)` returns the smaller of two values
- `max(x,y)` returns the larger of two values
- `fill_n(array, N, value)` fills `array[0]..array[N-1]` with `value`

.....

Building a Library of Array Templates

Second try, using templates:

- Header:

```
#ifndef ARRAYUTILS_H
#define ARRAYUTILS_H



//  Add to the end
```



```
//  - Assumes that we have a separate integer (size) indicating how
//     many elements are in the array
//  - and that the "true" size of the array is at least one larger
//      than the current value of that counter
template <typename T>
void addToEnd (T* array, int& size, T value)
{
   array[size] = value;
   ++size;
}



// Add value into array[index], shifting all elements already in positions
//    index..size-1 up one, to make room.
//  - Assumes that we have a separate integer (size) indicating how
//     many elements are in the array
//  - and that the "true" size of the array is at least one larger
//      than the current value of that counter

template <typename T>
void addElement (T* array, int& size, int index, T value)
{
  // Make room for the insertion
  int toBeMoved = size - 1;
  while (toBeMoved >= index) {
    array[toBeMoved+1] = array[toBeMoved];
    --toBeMoved;
  }
  // Insert the new value
  array[index] = value;
```


```
  ++size;
}


// Assume the elements of the array are already in order
// Find the position where value could be added to keep
//    everything in order, and insert it there.
// Return the position where it was inserted
//  - Assumes that we have a separate integer (size) indicating how
//     many elements are in the array
//  - and that the "true" size of the array is at least one larger
//      than the current value of that counter

template <typename T>
int addInOrder (T* array, int& size, T value)
{
  // Make room for the insertion
  int toBeMoved = size - 1;
  while (toBeMoved >= 0 && value < array[toBeMoved]) {
    array[toBeMoved+1] = array[toBeMoved];
    --toBeMoved;
  }
  // Insert the new value
  array[toBeMoved+1] = value;
  ++size;
  return toBeMoved+1;
}


// Search an array for a given value, returning the index where
//    found or -1 if not found.
```


```
template <typename T>
int seqSearch(const T list[], int listLength, T searchItem)
{
    int loc;

    for (loc = 0; loc < listLength; loc++)
        if (list[loc] == searchItem)
            return loc;

    return -1;
}


// Search an ordered array for a given value, returning the index where
//    found or -1 if not found.
template <typename T>
int seqOrderedSearch(const T list[], int listLength, T searchItem)
{
    int loc = 0;

    while (loc < listLength && list[loc] < searchItem)
      {
       ++loc;
      }
    if (loc < listLength && list[loc] == searchItem)
       return loc;
    else
       return -1;
}
```


```
// Removes an element from the indicated position in the array, moving
// all elements in higher positions down one to fill in the gap.
template <typename T>
void removeElement (T* array, int& size, int index)
{
  int toBeMoved = index + 1;
  while (toBeMoved < size) {
    array[toBeMoved] = array[toBeMoved+1];
    ++toBeMoved;
  }
  --size;
}



// Search an ordered array for a given value, returning the index where
//    found or -1 if not found.
template <typename T>
int binarySearch(const T list[], int listLength, T searchItem)
{
    int first = 0;
    int last = listLength - 1;
    int mid;

    bool found = false;

    while (first <= last && !found)
    {
        mid = (first + last) / 2;

        if (list[mid] == searchItem)
```


```
            found = true;
        else
            if (searchItem < list[mid])
                last = mid - 1;
            else
                first = mid + 1;
    }

    if (found)
        return mid;
    else
        return -1;
}




#endif
```

- Test driver:

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

#include "arrayUtils.h"

using namespace std;

void doTest (char** data, int size)
{
```


```
  string array[size];
  int aSize = 0;
  for (int i = 0; i < size; ++i)
    {
      addInOrder (array, aSize, string(data[i]));
    }
  for (int i = 0; i < size; ++i)
    {
      int index = seqOrderedSearch (array, aSize, string(data[i]));
      cout << data[i] << " was inserted at position "
       << index << endl;
    }

}


int main (int argc, char** argv)
{
  doTest (argv+1, argc-1);
  return 0;
}
```

.....

Function Templates Are Not Functions

Is it a problem that we have the bodies in a `.h` file?

- No, because function templates are not functions, they are *patterns* for functions.

. . . . . .

Patterns versus Instances

In the same way that

- This is not an admission application.
  - But this *is* an admission application.
- These are not functions.
  - The code generated by the compiler in response to our calls are the functions.

. . . . . .

4 Vectors

Keeping Information Together

One criticism of functions like

```
void addToEnd (std::string* array, int& size,
               std::string value);
int addInOrder (std::string* array, int& size,
    std::string value);
```

is that they separate the array, the size, and the capacity

- Easy for programmers to lose track of which integer counter applies to which array
- Complicates functions to pass this information separately.

.....

Wrapping arrays within structs

One solution: use a *struct* to gather the related elements together:

```
/// A collection of items
struct ItemSequence {
   static const int capacity = 500;
   int size;
   Item data[capacity];
};
```

.....

A Better Version

Using dynamic allocation, we can be more flexible about the capacity:

```
struct ItemSequence {
   int capacity;
   int size;
   Item* data;

   ItemSequence (int cap);
```


```
    void addToEnd (Item item);
};
```

.....

Implementing the Sequence

```
ItemSequence::ItemSequence (int cap)
    : capacity(cap), size(0)
{
  data = new Item[capacity];
}

void ItemSequence::addToEnd (Item item)
{
  if (size < capacity)
  {
    data[size] = item;
    ++size;
  }
  else
    cerr << "**Error: ItemSequence is full" << endl;
}
```

.....

This is, however, such a common pattern, that the designers of C++ had pity on us and did even better.

Vectors – the Un-Array

The *vector* is an array-like structure provided in the `std` header `<vector>`.

- Think of it as an array that can grow at the high end
- actually another kind of template, a *class template*

.....

Declaring Vectors

```
std::vector<int> vi; // a vector of 0 ints
std::vector<std::string> vs (10); // a vector of 10
                                  //   empty strings
std::vector<float> vf (5, 1.0); // a vector of 5
                                //    floats, all 1.0
```

- The type name inside the < > describes the elements contained inside the vector

.....

Accessing Elements in a Vector

Use the [ ] just as with an array:

```
vector<int> v(10);
for (int i = 0; i < 10; ++i)
  {
   int j;
   cin >> j;
   v[i] = j + 1;
   cout << v[i] << endl;
  }
```

.....

Size of a Vector

```
void foo (vector<int>& v) {
  for (int i = 0; i < v.size() ; ++i)
    {
     int j;
     cin >> j;
```


```
    v[i] = j + 1;
    cout << v[i] << endl;
   }
}
```

- Vectors remember their own size
- Accessed via `size()`
- With vectors, we do *not* over-allocate extra spaces in the vector and use a separate counter

. . . . . .

Adding to a Vector

Adding to the end:

- This is how we "grow" a vector.

```
v.push_back(x);
```

. . . . . .

Adding to the Middle

```
template <class Vector, class T>
void addElement (Vector& v, int index, T value)
{
  // Make room for the insertion
  int toBeMoved = v.size() - 1;
  v.push_back(value); // expand vector by 1 slot
  while (toBeMoved >= index) {
    v[toBeMoved+1] = v[toBeMoved];
    --toBeMoved;
  }
```


```
  // Insert the new value
  v[index] = value;
}
```

.....

Adding to the Middle: v2

Actually, *vector*s provide a built-in operation for inserting into the middle:

```
string* a; // array of strings
vector<string> v;
int k, n;
   ⋮
v.insert (v.begin()+k, "Hello");
addElement (a, n, k, "Hello");
```

The last two statements do the same thing.

- But `insert` is a built-in member function of *vector*s
- The way of specifying the position is a bit odd.
  - `v.begin()` is a "pointer" to the start of the vector
    - Like a is a pointer to the start of the array
  - `v.begin() + k` is a "pointer" to `v[k]`
    - Like $a + k$ is a pointer to the `a[k]`

.....

Add in Order to Vector

```
template <class Vector, class T>
int addInOrder (Vector& v, T value)
{
```


```
  // Make room for the insertion
  int toBeMoved = v.size() - 1;
  v.push_back(value); // expand vector by 1 slot
  while (toBeMoved >= 0 && value < v[toBeMoved]) {
    v[toBeMoved+1] = v[toBeMoved];
    --toBeMoved;
  }
  // Insert the new value
  v[toBeMoved+1] = value;
  return toBeMoved+1;
}
```

.....

4.0.1 Vectors versus Arrays

Advantages of Vectors

- Can grow as necessary
- Need not worry about pointers, allocation, delete
- Vectors can copy: `v1 = v2;`

.....

Disadvantages of Vectors

- A bit slower than arrays
  - `push_back` is sometimes very slow
- Can waste a lot of storage
  - but so can arrays if we have to guess at max

- Harder to work with in a debugger

. . . . . .

5 Multi-Dimension Arrays

Multi-Dimension Arrays

- Used in situations where we would arrange data into a table rather than a list.
- If we know how many rows and columns,
  Allocate array as

```
int array[numRows][numCols];
```

  – Access as `array[i][j]`

0,0	0,1	0,2
1,0	1,1	1,2
2,0	2,1	2,2
3,0	3,1	3,2

. . . . . .

Example

```
#include <iostream>

using namespace std;
```


```
const int NumRows = 4;
const int NumCols = 3;

int main (int argc, char** argv)
{
  double table[NumRows][NumCols];
  for (int row = 0; row < NumRows; ++row)
    for (int col = 0; col < NumCols; ++col)
      table[row][col] = 1.0 / (1.0 + row + col);

  for (int row = 0; row < NumRows; ++row)
    for (int col = 0; col < NumCols; ++col)
      cout << row << " " << col << " "
       << table[row][col] << endl;


  return 0;
}
```

5.1 Arrays of Arrays

Arrays of Arrays

- If we do not know (at compile time) how many rows and columns,
  - Declare array as

    ```
    int** array;
    ```

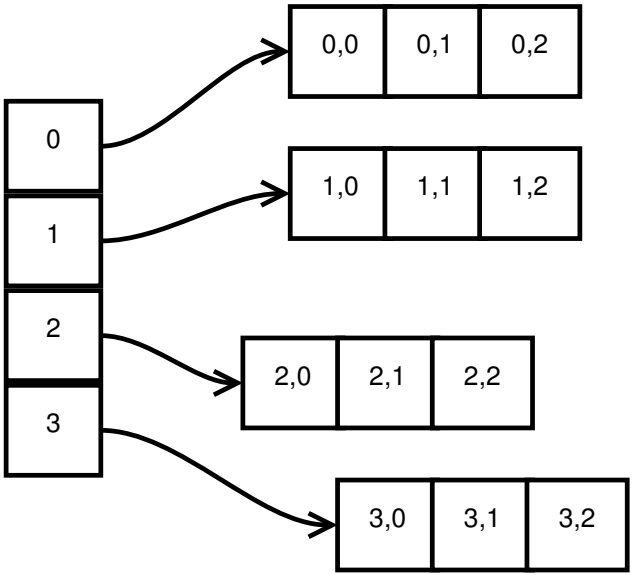
  - Allocate as an array of pointers:

    ```
    array = new int*[numRows];
    ```

  - Each row must then be allocated as an array

    ```
    array[i] = new int[numCols];
    ```

  - Access as `array[i][j]`
- Example:

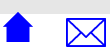



```
#include <iostream>

using namespace std;

const int NumCols = 3;

int main (int argc, char** argv)
{
  int NumCols;
  int NumRows;
  cout << "How many rows? " << flush;
  cin >> NumRows;
  cout << "How many columns? " << flush;
```

```
  cin >> NumCols;

  double** table = new double*[NumRows];
  for (int row = 0; row < NumRows; ++row)
    {
      table[row] = new double[NumCols];
      for (int col = 0; col < NumCols; ++col)
    table[row][col] = 1.0 + row + col;
    }
  for (int row = 0; row < NumRows; ++row)
    for (int col = 0; col < NumCols; ++col)
      cout << row << " " << col << " "
       << table[row][col] << endl;

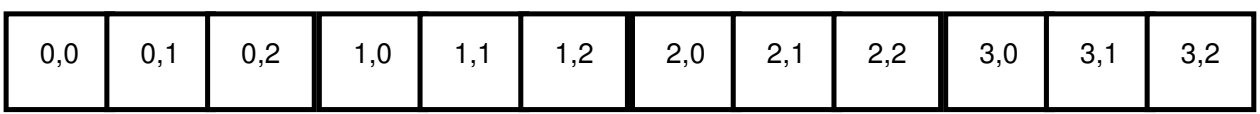

  return 0;
}
```

.....

Linearized Arrays

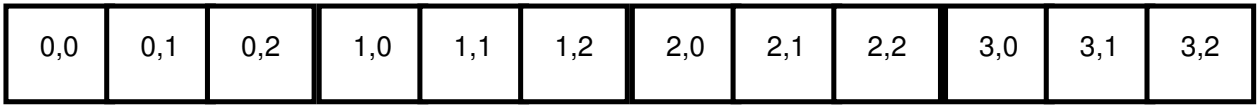
We can map 2-D arrays onto a linear structure

```
int index (int i, j, numCols)
{
  return j + i * numColss;
}
```



.....

Linearized Array Code



- Declare array as

```
int* array;
```

- Allocate as a 1-D array

```
array = new int[numRows*numCols];
```

- Access as `array[index(i,j,numCols)]`