Arrays

CS 115

Dr. Joseph Eremondi, adapted from Dr. Shakil Khan, Dr. Philip Fong, and Dr. Howard Hamilton

Last updated: January 7, 2025

One, two, and multi-dimensional arrays

Motivation

- Print 1000 numbers in reverse order

Motivation

- Print 1000 numbers in reverse order

Motivation

- Print 1000 numbers in reverse order

```
int value0;
int value1;
int value2;
// ...
int value999;

cin >> value0;
cin >> value1;
// ...
cin >> value999;

cout << value999 << endl;
cout << value998 << endl;
// ...
cout << value0 << endl;
```

Motivation (cont'd)

- How about 1000000 numbers?

Motivation (cont'd)

- How about 1000000 numbers?
- Tedious, not scalable, and error prone

Motivation (cont'd)

- How about 1000000 numbers?
- Tedious, not scalable, and error prone
- Solution: use aggregate data type

Motivation (cont'd)

- How about 1000000 numbers?
- Tedious, not scalable, and error prone
- Solution: use aggregate data type
  - homogenous components

Motivation (cont'd)

- How about 1000000 numbers?
- Tedious, not scalable, and error prone
- Solution: use aggregate data type
  - homogenous components
  - indexing support

Motivation (cont'd)

- How about 1000000 numbers?
- Tedious, not scalable, and error prone
- Solution: use aggregate data type
  - homogenous components
  - indexing support
  - constant time access

Motivation (cont'd)

- How about 1000000 numbers?
- Tedious, not scalable, and error prone
- Solution: use aggregate data type
  - homogenous components
  - indexing support
  - constant time access
  - random access

Motivation (cont'd)

- How about 1000000 numbers?
- Tedious, not scalable, and error prone
- Solution: use aggregate data type
  - homogenous components
  - indexing support
  - constant time access
  - random access

Motivation (cont'd)

- How about 1000000 numbers?
- Tedious, not scalable, and error prone
- Solution: use aggregate data type
  - homogenous components
  - indexing support
  - constant time access
  - random access

```
int a[120000];    // Array declaration

for (int i = 0; i < 120000; i++)
  cin >> a[i];    // Array access
for (int i = 119999; i >= 0; i--)
  cout << a[i] << endl;
```

Array Operations

- Call the things we store in the array *elements*

Array Operations

- Call the things we store in the array *elements*
- Get the ith element's value: `array[i]`

Array Operations

- Call the things we store in the array *elements*
- Get the ith element's value: `array[i]`
- Set the ith element: `array[i] = someValue;`

Simple arrays

Simple arrays

```
const int N = 120000;
int a[N];    // Array declaration

for (int i = 0; i < N; i++)
  cin >> a[i];    // Array access
for (int i = N-1; i >= 0; i--)
  cout << a[i] << endl;
```

- Array size must be a constant expression

Simple arrays

```
const int N = 120000;
int a[N];    // Array declaration

for (int i = 0; i < N; i++)
  cin >> a[i];    // Array access
for (int i = N-1; i >= 0; i--)
  cout << a[i] << endl;
```

- Array size must be a constant expression
- Easy to change size: just update N (the rest of the program remains intact)

Passing arrays as arguments

Passing arrays as arguments

```
int sumArray(int a[], unsigned int n) // Array argument
{
  int sum = 0;
  for (int i = 0; i < n; i++)
    sum += a[i];
  return sum;
}

int main()
{
  // Array initialization
  int a[] = { 3, 24, -88, 17, -1 };
  cout << sumArray(a, 5) << endl;
}
```

- Array size can be left unspecified in array initialization syntax

Passing arrays as arguments

- Array arguments are always automatically passed by reference

Passing arrays as arguments

- Array arguments are always automatically passed by reference
- no special notation is require

Passing arrays as arguments

- Array arguments are always automatically passed by reference
- no special notation is require

Passing arrays as arguments

- Array arguments are always automatically passed by reference
- no special notation is require

```
// int sumArray(int& a[], unsigned int n) - INCORRECT
int sumArray(int a[], unsigned int n)    // CORRECT
{
  ...
    }
```

- Works for arrays of all sizes (size is passed as a separate argument)

Passing arrays as arguments

- Array arguments are always automatically passed by reference
- no special notation is require

```
// int sumArray(int& a[], unsigned int n) - INCORRECT
int sumArray(int a[], unsigned int n)    // CORRECT
{
  ...
    }
```

- Works for arrays of all sizes (size is passed as a separate argument)
- Interface not safe: can modify the content of A

A Safer Interface

A Safer Interface

```
int sumArray(int a[], unsigned int n)
// not safe, sumArray can modify A!
```

- Use the following instead:

A Safer Interface

```
int sumArray(int a[], unsigned int n)
// not safe, sumArray can modify A!
```

- Use the following instead:

A Safer Interface

```
int sumArray(int a[], unsigned int n)
// not safe, sumArray can modify A!
```

- Use the following instead:

```
int sumArray(const int a[], unsigned int n)
```

- How to figure out array size when passing n if the size was left unspecified when declaring it?

A Safer Interface

```
int sumArray(int a[], unsigned int n)
// not safe, sumArray can modify A!
```

- Use the following instead:

```
int sumArray(const int a[], unsigned int n)
```

- How to figure out array size when passing n if the size was left unspecified when declaring it?
- use sizeof function:

A Safer Interface

```
int sumArray(int a[], unsigned int n)
// not safe, sumArray can modify A!
```

- Use the following instead:

```
int sumArray(const int a[], unsigned int n)
```

- How to figure out array size when passing n if the size was left unspecified when declaring it?
- use sizeof function:

A Safer Interface

```
int sumArray(int a[], unsigned int n)
// not safe, sumArray can modify A!
```

- Use the following instead:

```
int sumArray(const int a[], unsigned int n)
```

- How to figure out array size when passing n if the size was left unspecified when declaring it?
- use sizeof function:

```
int a[] = {1,2,6,3,8};
int x = sumArray(a, sizeof(a) / sizeof(int));
```

Play time

- Check if integer array sorted

Play time

- Check if integer array sorted

Play time

- Check if integer array sorted

```
bool arrayIsSorted(const int a[], unsigned int n){
  for (int i = 0; i < n-1; i++){
    if (a[i] > a[i+1])
      return false;
  }
  return true;
}
```

Play time

- Reversing items in integer array

Play time

- Reversing items in integer array

Play time

- Reversing items in integer array

```
void swap(int &a, int &b) {
  int tmp = a;
  a = b;
  b = tmp;
}
// below a[] is not a constant as want to produce side-effect
void reverseArray(int a[], unsigned int n) {
  for (int i = 0; i < n/2; i++)
    swap(a[i], a[n - i - 1]);
}
```

Processing subarrays

- Compute the sum of an array segment

Processing subarrays

- Compute the sum of an array segment

Processing subarrays

- Compute the sum of an array segment

```
// pos   : index of the first component in the subarray
// count: total number of components in the subarray
int sumSubarray(const int a[],
                unsigned int pos,
                unsigned int count){
  int sum = 0;
  for (int i = pos; i < pos + count; i++)
    sum += a[i];

  return sum;
}
```

Processing subarrays

- Another way to do the same thing

Processing subarrays

- Another way to do the same thing

Processing subarrays

- Another way to do the same thing

```
// begin: index of first component in the subarray
// end   : index of the last component in the subarray
int sumSubarray(const int a[],
                unsigned int begin,
                unsigned int end){
  assert(begin <= end);
  int sum = 0;
  for (int i = begin; i <= end; i++)
    sum += a[i];

  return sum;
}
```

Subtleties

- C++ does not check if array indices are within bound

Subtleties

- C++ does not check if array indices are within bound
- it’s your responsibility

Subtleties

- C++ does not check if array indices are within bound
- it’s your responsibility
- Array Copying

Subtleties

- C++ does not check if array indices are within bound
- it’s your responsibility
- Array Copying

Subtleties

- C++ does not check if array indices are within bound
- it’s your responsibility
- Array Copying

```
a = b // invalid
```

- copy cell by cell:

Subtleties

- C++ does not check if array indices are within bound
- it’s your responsibility
- Array Copying

```
a = b // invalid
```

- copy cell by cell:

Subtleties

- C++ does not check if array indices are within bound
- it’s your responsibility
- Array Copying

```
a = b // invalid
```

- copy cell by cell:

```
a[6]=b[9] // works!
```

Subtleties

- Array Comparison

Subtleties

- Array Comparison

Subtleties

- Array Comparison

```
if(a == b) // invalid
```

- compare each pair of cells at a time

Subtleties

- Array Comparison

```
if(a == b) // invalid
```

- compare each pair of cells at a time
- No need to return array as function output, uses call by reference anyway!

Prof's Aside

- C++ arrays are *unsafe*

Prof's Aside

- C++ arrays are *unsafe*
- This is *terrible* language design

Prof's Aside

- C++ arrays are *unsafe*
- This is *terrible* language design
  - Billions of dollars and many security incidents caused by unsafe memory access

Prof's Aside

- C++ arrays are *unsafe*
- This is *terrible* language design
  - Billions of dollars and many security incidents caused by unsafe memory access
  - Error cost outweighs performance cost of checking array bounds

Prof's Aside

- C++ arrays are *unsafe*
- This is *terrible* language design
  - Billions of dollars and many security incidents caused by unsafe memory access
  - Error cost outweighs performance cost of checking array bounds
  - Most checks can be optimized out by the compiler

Prof's Aside

- C++ arrays are *unsafe*
- This is *terrible* language design
  - Billions of dollars and many security incidents caused by unsafe memory access
  - Error cost outweighs performance cost of checking array bounds
  - Most checks can be optimized out by the compiler
- C++ will never change

Prof's Aside

- C++ arrays are *unsafe*
- This is *terrible* language design
  - Billions of dollars and many security incidents caused by unsafe memory access
  - Error cost outweighs performance cost of checking array bounds
  - Most checks can be optimized out by the compiler
- C++ will never change
  - Backwards compatibility

Prof's Aside

- C++ arrays are *unsafe*
- This is *terrible* language design
  - Billions of dollars and many security incidents caused by unsafe memory access
  - Error cost outweighs performance cost of checking array bounds
  - Most checks can be optimized out by the compiler
- C++ will never change
  - Backwards compatibility
  - `std::array` is safe but isn't the default

Prof's Aside

- C++ arrays are *unsafe*
- This is *terrible* language design
  - Billions of dollars and many security incidents caused by unsafe memory access
  - Error cost outweighs performance cost of checking array bounds
  - Most checks can be optimized out by the compiler
- C++ will never change
  - Backwards compatibility
  - `std::array` is safe but isn't the default
- Languages like Rust make sure that these errors are *impossible*

Prof's Aside

- C++ arrays are *unsafe*
- This is *terrible* language design
  - Billions of dollars and many security incidents caused by unsafe memory access
  - Error cost outweighs performance cost of checking array bounds
  - Most checks can be optimized out by the compiler
- C++ will never change
  - Backwards compatibility
  - `std::array` is safe but isn't the default
- Languages like Rust make sure that these errors are *impossible*
  - Unless you explicitly disable safety

Example

Example

```
#include <iostream>
using namespace std;
int main(){
  char passwd[8] = "secret";
  char username[8] = "bob101";
  string toPrint = "";
  // Oops reading past end of array!
  for (int i = 0; i < 16; i++){
    toPrint += username[i];
  }
  cout << toPrint << endl;
}
```

Example

```
#include <iostream>
using namespace std;
int main(){
  char passwd[8] = "secret";
  char username[8] = "bob101";
  string toPrint = "";
  // Oops reading past end of array!
  for (int i = 0; i < 16; i++){
    toPrint += username[i];
  }
  cout << toPrint << endl;
}
```

```
bob101secret
```

Two Dimensional Arrays

Motivation

- Want to store quantity of different products sold in a store

Motivation

- Want to store quantity of different products sold in a store
- but for multiple locations/regions

Motivation

- Want to store quantity of different products sold in a store
- but for multiple locations/regions
- Conceptually can store as a matrix, where rows represent different locations and columns represent different products

Motivation

- Want to store quantity of different products sold in a store
- but for multiple locations/regions
- Conceptually can store as a matrix, where rows represent different locations and columns represent different products
- `sales[2][1]` are the total number of items sold for location 2 and product 1

Motivation

- Want to store quantity of different products sold in a store
- but for multiple locations/regions
- Conceptually can store as a matrix, where rows represent different locations and columns represent different products
- `sales[2][1]` are the total number of items sold for location 2 and product 1
- recall item n is the (n+1)-th item

Motivation

- Want to store quantity of different products sold in a store
- but for multiple locations/regions
- Conceptually can store as a matrix, where rows represent different locations and columns represent different products
- `sales[2][1]` are the total number of items sold for location 2 and product 1
- recall item n is the (n+1)-th item
  - index starts from 0!

Declaration and Access

Declaration and Access

```
const unsigned int NUM_OF_REGIONS = 4;
const unsigned int NUM_OF_PRODUCTS = 3;

unsigned int sales[NUM_OF_REGIONS][NUM_OF_PRODUCTS];
```

- To access sales figure for first product in second region, use:

Declaration and Access

```
const unsigned int NUM_OF_REGIONS = 4;
const unsigned int NUM_OF_PRODUCTS = 3;

unsigned int sales[NUM_OF_REGIONS][NUM_OF_PRODUCTS];
```

- To access sales figure for first product in second region, use:

Declaration and Access

```
const unsigned int NUM_OF_REGIONS = 4;
const unsigned int NUM_OF_PRODUCTS = 3;

unsigned int sales[NUM_OF_REGIONS][NUM_OF_PRODUCTS];
```

- To access sales figure for first product in second region, use:

```
sales[1][0] // recall, indices start from 0
```

- e.g., want to set sales figure for first product in second region to 500

Declaration and Access

```
const unsigned int NUM_OF_REGIONS = 4;
const unsigned int NUM_OF_PRODUCTS = 3;

unsigned int sales[NUM_OF_REGIONS][NUM_OF_PRODUCTS];
```

- To access sales figure for first product in second region, use:

```
sales[1][0] // recall, indices start from 0
```

- e.g., want to set sales figure for first product in second region to 500

Declaration and Access

```
const unsigned int NUM_OF_REGIONS = 4;
const unsigned int NUM_OF_PRODUCTS = 3;

unsigned int sales[NUM_OF_REGIONS][NUM_OF_PRODUCTS];
```

- To access sales figure for first product in second region, use:

```
sales[1][0] // recall, indices start from 0
```

- e.g., want to set sales figure for first product in second region to 500

```
sales[1][0] = 500;
```

Populating and Accessing

Populating and Accessing

```
// Read input stream
for (unsigned int region = 0; region < NUM_OF_REGIONS; region++)
  for (unsigned int product = 0; product < NUM_OF_PRODUCTS; product++)
    cin >> sales[region][product];

// total sales for a particular product (product 0)
unsigned int total_sales = 0;

for (unsigned int region = 0; region < NUM_OF_REGIONS; region++)
  // add up sales from all regions for product 0
  total_sales += sales[region][0];
```

- Can you compute total sales from region 1?

Passing 2D Arrays

Passing 2D Arrays

```
unsigned int sumProductSales(
        unsigned int sales[NUM_OF_REGIONS][NUM_OF_PRODUCTS],
        unsigned int product)
{
  unsigned int total_sales = 0;
  for (unsigned int region = 0; region < NUM_OF_REGIONS; region++)
    total_sales += sales[region][product];

  return total_sales;
}
```

- Can you implement a safer interface?

Passing 2D Arrays

```
unsigned int sumProductSales(
        unsigned int sales[NUM_OF_REGIONS][NUM_OF_PRODUCTS],
        unsigned int product)
{
  unsigned int total_sales = 0;
  for (unsigned int region = 0; region < NUM_OF_REGIONS; region++)
    total_sales += sales[region][product];

  return total_sales;
}
```

- Can you implement a safer interface?
- As usual, can leave size of first dimension unspecified, e.g. `int F(int arr[ ][SIZE])`

Passing 2D Arrays

```
unsigned int sumProductSales(
        unsigned int sales[NUM_OF_REGIONS][NUM_OF_PRODUCTS],
        unsigned int product)
{
  unsigned int total_sales = 0;
  for (unsigned int region = 0; region < NUM_OF_REGIONS; region++)
    total_sales += sales[region][product];

  return total_sales;
}
```

- Can you implement a safer interface?
- As usual, can leave size of first dimension unspecified, e.g. `int F(int arr[ ][SIZE])`
- but not the second one (why?)

Making things more modular

- So we can change internal representation without changing interface

Making things more modular

- So we can change internal representation without changing interface

Making things more modular

- So we can change internal representation without changing interface

```
// Implement a function that returns
// the value of one element from the sales array
unsigned int getSales(
        const unsigned int sales[NUM_OF_REGIONS][NUM_OF_PRODUCTS],
        unsigned int r, unsigned int p){
  return sales[r][p];
}
// Implement a function that sets the value
// of one element from the sales array
void setSales(unsigned int sales[NUM_OF_REGIONS][NUM_OF_PRODUCTS],
              unsigned int r, unsigned int p, unsigned int v){
  sales[r][p] = v;
}
```

Using typedef

Using typedef

```
// too lazy to write long types? Use typedef instead!

typedef unsigned int Sales[NUM_OF_REGIONS][NUM_OF_PRODUCTS];

unsigned int sumSales(const Sales sales){
  ...
    }
```

Simulating Two-dimensional Arrays by One-dimensional Ones

Simulating Two-dimensional Arrays by One-dimensional Ones

```
unsigned int sales[NUM_OF_REGIONS][NUM_OF_PRODUCTS];
```

- versus

Simulating Two-dimensional Arrays by One-dimensional Ones

```
unsigned int sales[NUM_OF_REGIONS][NUM_OF_PRODUCTS];
```

- versus

Simulating Two-dimensional Arrays by One-dimensional Ones

```
unsigned int sales[NUM_OF_REGIONS][NUM_OF_PRODUCTS];
```

- versus

```
unsigned int _sales[NUM_OF_REGIONS * NUM_OF_PRODUCTS];
```

- Issue: how to map between these two?

Simulating Two-dimensional Arrays by One-dimensional Ones

```
unsigned int sales[NUM_OF_REGIONS][NUM_OF_PRODUCTS];
```

- versus

```
unsigned int _sales[NUM_OF_REGIONS * NUM_OF_PRODUCTS];
```

- Issue: how to map between these two?
  - row-major vs. column-major order

Simulating Two-dimensional Arrays by One-dimensional Ones

```
unsigned int sales[NUM_OF_REGIONS][NUM_OF_PRODUCTS];
```

- versus

```
unsigned int _sales[NUM_OF_REGIONS * NUM_OF_PRODUCTS];
```

- Issue: how to map between these two?
  - row-major vs. column-major order
  - e.g. `sales[i][j]`

Simulating Two-dimensional Arrays by One-dimensional Ones

```
unsigned int sales[NUM_OF_REGIONS][NUM_OF_PRODUCTS];
```

- versus

```
unsigned int _sales[NUM_OF_REGIONS * NUM_OF_PRODUCTS];
```

- Issue: how to map between these two?
  - row-major vs. column-major order
  - e.g. `sales[i][j]`
    - same as `_sales[i * NUM_OF_PRODUCTS + j]` in row-major

Simulating Two-dimensional Arrays by One-dimensional Ones

```
unsigned int sales[NUM_OF_REGIONS][NUM_OF_PRODUCTS];
```

- versus

```
unsigned int _sales[NUM_OF_REGIONS * NUM_OF_PRODUCTS];
```

- Issue: how to map between these two?
  - row-major vs. column-major order
  - e.g. `sales[i][j]`
    - same as `_sales[i * NUM_OF_PRODUCTS + j]` in row-major
- Now you know why the size of the 2nd dimension can't be left unspecified!

Simulating Two-dimensional Arrays by One-dimensional Ones

```
unsigned int sales[NUM_OF_REGIONS][NUM_OF_PRODUCTS];
```

- versus

```
unsigned int _sales[NUM_OF_REGIONS * NUM_OF_PRODUCTS];
```

- Issue: how to map between these two?
  - row-major vs. column-major order
  - e.g. `sales[i][j]`
    - same as `_sales[i * NUM_OF_PRODUCTS + j]` in row-major
- Now you know why the size of the 2nd dimension can't be left unspecified!
  - Can you write the formula for column-major order?

Using Row-Major Order

Using Row-Major Order

```
unsigned int totalSales = 0;

for (unsigned int region = 0; region < NUM_OF_REGIONS; region++)
  for (unsigned int product = 0;
       product < NUM_OF_PRODUCTS;
       product++){
    totalSales += _sales[region * NUM_OF_PRODUCTS + product];
  }
```

- This is why we need to know the size of the second dimension

Using Row-Major Order

```
unsigned int totalSales = 0;

for (unsigned int region = 0; region < NUM_OF_REGIONS; region++)
  for (unsigned int product = 0;
       product < NUM_OF_PRODUCTS;
       product++){
    totalSales += _sales[region * NUM_OF_PRODUCTS + product];
  }
```

- This is why we need to know the size of the second dimension
  - To calculate offset

Multi-dimensional Arrays

Multi-dimensional Arrays

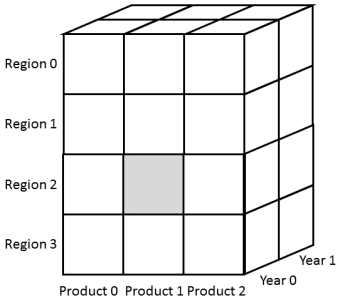
```
const unsigned int NUM_YEARS = 2;
const unsigned int NUM_REGIONS = 4;
const unsigned int NUM_PRODUCTS = 3;

typedef unsigned int Sales[NUM_YEARS][NUM_REGIONS][NUM_PRODUCTS];

unsigned int total_sales = 0;
for (unsigned int year = 0; year < NUM_YEARS; year++)
  for (unsigned int region = 0; region < NUM_REGIONS; region++)
    for (unsigned int product = 0; product < NUM_PRODUCTS; product++)
      total_sales += sales[year][region][product];
```

Simulating 3d with 1d

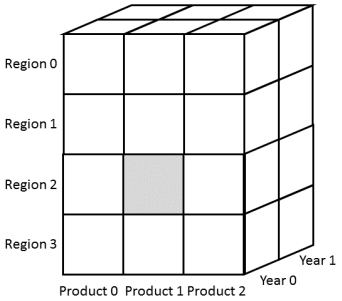
- `Sales[year][region][product]`



Mem-Pos	
0	Y0R0P0
1	Y0R0P1
2	Y0R0P2
3	Y0R1P0
4	Y0R1P1
5	Y0R1P2
6	Y0R2P0
7	Y0R2P1
8	Y0R2P2
9	Y0R3P0
10	Y0R3P1
11	Y0R3P2
12	Y1R0P0
13	Y1R0P1
14	Y1R0P2
15	Y1R1P0
16	Y1R1P1
17	Y1R1P2
18	Y1R2P0
19	Y1R2P1
20	Y1R2P2
21	Y1R3P0
22	Y1R3P1
23	Y1R3P2

Simulating 3d with 1d

- `Sales[year][region][product]`
- vs `_Sales[(year * NUM_REGS * NUM_PRODS) + (region * NUM_OF_PRODS) + product]`



Mem-Pos	
0	Y0R0P0
1	Y0R0P1
2	Y0R0P2
3	Y0R1P0
4	Y0R1P1
5	Y0R1P2
6	Y0R2P0
7	Y0R2P1
8	Y0R2P2
9	Y0R3P0
10	Y0R3P1
11	Y0R3P2
12	Y1R0P0
13	Y1R0P1
14	Y1R0P2
15	Y1R1P0
16	Y1R1P1
17	Y1R1P2
18	Y1R2P0
19	Y1R2P1
20	Y1R2P2
21	Y1R3P0
22	Y1R3P1
23	Y1R3P2

Simulating Multi-dimensional Arrays

- In general for a d-dimensional array with dimensions S_1, S_2, …, S_d, the element at `Item[n_1][n_2]...[n_d]` can be represented as a single dimensional array with the following index

Simulating Multi-dimensional Arrays

- In general for a d-dimensional array with dimensions S_1, S_2, …, S_d, the element at `Item[n_1][n_2]...[n_d]` can be represented as a single dimensional array with the following index

Simulating Multi-dimensional Arrays

- In general for a d-dimensional array with dimensions S_1, S_2, …, S_d, the element at `Item[n_1][n_2]...[n_d]` can be represented as a single dimensional array with the following index

```
_Item[n_d + S_d * (n_{d-1} + S_{d-1}
   * (n_{d-2} + S_{d-2} * (...+S_2*n_1) ... ))]
```