Arrays

CS 115

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

Last updated: December 20, 2024

One, two, and multi-dimensional arrays

Motivation

Motivation

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

cin >> value0;
cin >> value1;
cin >> value2;
// ...
cin >> value999;

cout << value999 << endl;
cout << value998 << endl;
cout << value997 << endl;
// ...
cout << value0 << endl;
```

Motivation (cont’d)

- Tedious, not scalable, and error prone

Motivation (cont'd)

- Tedious, not scalable, and error prone
- Solution: use aggregate data type

Motivation (cont'd)

- Tedious, not scalable, and error prone
- Solution: use aggregate data type
    - homogenous components

Motivation (cont'd)

- Tedious, not scalable, and error prone
- Solution: use aggregate data type
    - homogenous components
    - indexing support

Motivation (cont'd)

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

Motivation (cont’d)

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

Motivation (cont'd)

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

Motivation (cont'd)

- 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;
```

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

```cpp
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

Passing arrays as arguments

Passing arrays as arguments

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

- Use the following instead:

Passing arrays as arguments

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

- Use the following instead:

Passing arrays as arguments

```
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?

Passing arrays as arguments

```
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:

Passing arrays as arguments

```
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:

Passing arrays as arguments

```
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

Play time

```
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

Play time

```
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

Processing subarrays

```
// 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

Processing subarrays

```
// 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!

Joey’s Aside

- C++ arrays are *unsafe*

Joey’s Aside

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

Joey’s Aside

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

Joey’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

Joey’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

Joey'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

Joey'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

Joey’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

Joey’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*

Joey'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

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

Two dimensional arrays

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

Two dimensional arrays

- 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

Two dimensional arrays

- 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

Two dimensional arrays

- 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 as index starts from `0`!

Two dimensional arrays

Two dimensional arrays

```
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:

Two dimensional arrays

```
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:

Two dimensional arrays

```
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

Two dimensional arrays

```
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

Two dimensional arrays

```
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;
```

Two dimensional arrays

Two dimensional arrays

```
// 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?

Two dimensional arrays

Two dimensional arrays

```c
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? (see slide 7)

Two dimensional arrays

```c
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? (see slide 7)
- As usual, can leave size of first dimension unspecified, e.g. int F(int arr[ ][SIZE])

Two dimensional 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? (see slide 7)
- 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

Making things more modular

```
// 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

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

- versus

```c
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

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

- versus

```c
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

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

- versus

```c
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?

Simulating Two-dimensional Arrays by One-dimensional Ones

Simulating Two-dimensional Arrays by One-dimensional Ones

```
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];
  }
```

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];
```

Multi-dimensional Arrays

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

Multi-dimensional Arrays

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

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

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

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) ... ))]
```