Arrays

Class 25

Array Elements as Function Parameters

- we have seen that array elements are simple variables
- they can be used anywhere "normal" variables can

```
unsigned values[] {10, 15, 20};
unsigned quotient;
unsigned remainder;
...
divide(values[2], values[0], quotient, remainder);
```

- the first two parameters are pass by value
- any changes made to them in the function do not affect the array, because the values are copied into the function

Arrays as Function Parameters

- recall:
  - an array name is the starting address of the array in memory
  - an array cannot be copied in one step, but only one element at a time

Arrays as Function Parameters

- recall:
  - an array name is the starting address of the array in memory
  - an array cannot be copied in one step, but only one element at a time
- thus an array cannot be passed by value into a function
- arrays can only be passed by reference into a function

Arrays as Function Parameters

- recall:
  - an array name is the starting address of the array in memory
  - an array cannot be copied in one step, but only one element at a time
- thus an array cannot be passed by value into a function
- arrays can only be passed by reference into a function
- because an array's name is simply the address of the array, it already is a reference (it's actually a pointer; we'll discuss the difference a bit later)

```
1  // from Gaddis program 7-17
2  void show_values(int values[], unsigned size);
3
4  int main()
5  {
6    const unsigned SIZE = 6;
7    int numbers[SIZE] {5, 10, 15, 20, 25, 30};
8
9    show_values(numbers, SIZE);
10   return 0;
11 }
12
13 void show_values (int values[], unsigned size)
14 {
15   for (unsigned index = 0; index < size; index++)
16   {
17     cout << values[index] << ' ';
18   }
19   cout << endl;
20 }
```

Arrays as Function Parameters

- several important things to note
  - lines 2 and 13: instead of an ampersand & to denote a reference parameter, we use empty square brackets to denote this is an array parameter
  - an array parameter works like a reference parameter (even though technically it's not quite the same)

Arrays as Function Parameters

- several important things to note
  - lines 2 and 13: instead of an ampersand & to denote a reference parameter, we use empty square brackets to denote this is an array parameter
  - an array parameter works like a reference parameter (even though technically it's not quite the same)

  - lines 2 and 13: we must pass the size of the array to the function because otherwise the function cannot determine the array size

Arrays as Function Parameters

- several important things to note
  - lines 2 and 13: instead of an ampersand & to denote a reference parameter, we use empty square brackets to denote this is an array parameter
  - an array parameter works like a reference parameter (even though technically it's not quite the same)

  - lines 2 and 13: we must pass the size of the array to the function because otherwise the function cannot determine the array size

  - line 15: based on what we know, we should use a foreach loop here
  - we cannot, because a foreach loop only works in the scope where the array was declared
  - this is because no other scope knows the array's size, only the scope where it was declared

Arrays as Function Parameters

- an array passed to a function works like a reference parameter
- pass by reference, not pass by value
- therefore, a change made to the array inside a function does affect the array in the calling scope

see program 7-19

`const` Array Parameters

- the fact that you cannot pass an array by value is problematic
- for "normal" variables, if you do not want a function to change their value, you use pass by value
- this prevents any change from affecting the calling scope

`const` Array Parameters

- the fact that you cannot pass an array by value is problematic
- for "normal" variables, if you do not want a function to change their value, you use pass by value
- this prevents any change from affecting the calling scope

- but you cannot pass an array by value
- how do you prevent the function from being able to alter the array?

`const` Array Parameters

- the fact that you cannot pass an array by value is problematic
- for "normal" variables, if you do not want a function to change their value, you use pass by value
- this prevents any change from affecting the calling scope

- but you cannot pass an array by value
- how do you prevent the function from being able to alter the array?

- you declare the array as a const array parameter
- always declare an array parameter const if the function will not change the array

```
1  // from Gaddis program 7-17, modified to use a const
2  void show_values(const int values[], unsigned size);
3
4  int main()
5  {
6    const unsigned SIZE = 6;
7    int numbers[SIZE] {5, 10, 15, 20, 25, 30};
8
9    show_values(numbers, SIZE);
10   return 0;
11 }
12
13 void show_values(const int values[], unsigned size)
14 {
15   for (unsigned index = 0; index < size; index++)
16   {
17     cout << values[index] << ' ';
18   }
19   cout << endl;
20 }
```

Tabular Data

- many human activities involve tabular arrangements of data

Tabular Data

- many human activities involve tabular arrangements of data
- imagine you are writing a program to help you plan your study abroad semester in Europe
- you need a table of distances among major cities

Tabular Data

- many human activities involve tabular arrangements of data
- imagine you are writing a program to help you plan your study abroad semester in Europe
- you need a table of distances among major cities

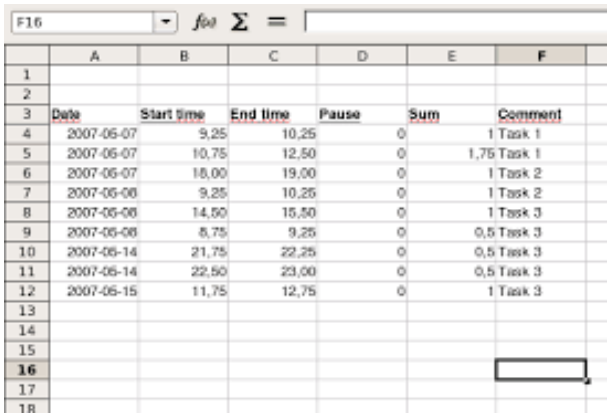
	Amster	Berlin	London	Madrid	Paris	Rome	Stock
Amster	0	648	494	1752	495	1735	1417
Berlin	648	0	1101	2349	1092	1518	1032
London	494	1101	0	1661	404	1870	1807
Madrid	1752	2349	1661	0	1257	2001	3138
Paris	495	1092	404	1257	0	1466	1881
Rome	1735	1588	1870	2001	1466	0	2620
Stock	1417	1032	1807	3138	1881	2620	0

Concept

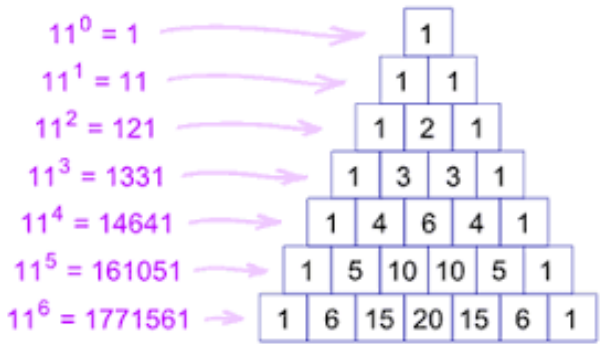
a 2-D arrangement of cells is a familiar concept

- mathematical matrix
- spreadsheet
- Pascal's triangle
- chessboard
- atoms in a lattice

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$



	A	B	C	D	E	F
1						
2						
3	Date	Start time	End time	Pause	Sum	Comment
4	2007-06-07	9,25	10,25	0	1	Task 1
5	2007-06-07	10,75	12,50	0	1,75	Task 1
6	2007-06-07	18,00	19,00	0	1	Task 2
7	2007-06-08	9,25	10,25	0	1	Task 2
8	2007-06-08	14,50	15,50	0	1	Task 3
9	2007-06-08	8,75	9,25	0	0,5	Task 3
10	2007-06-14	21,75	22,25	0	0,5	Task 3
11	2007-06-14	22,50	23,00	0	0,5	Task 3
12	2007-06-15	11,75	12,75	0	1	Task 3
13						
14						
15						
16						
17						
18						



all these can be modeled in a program with a 2-dimensional array structure

2-D Array

- a set of parallel arrays is not appropriate for this data
- instead, we need a two-dimensional array

2-D Array

- a set of parallel arrays is not appropriate for this data
- instead, we need a two-dimensional array
- C++ allows this

```
const unsigned NUMBER_OF_CITIES = 7;
unsigned distances[NUMBER_OF_CITIES][NUMBER_OF_CITIES];
```

2-D Array

- a set of parallel arrays is not appropriate for this data
- instead, we need a two-dimensional array
- C++ allows this

```
const unsigned NUMBER_OF_CITIES = 7;
unsigned distances[NUMBER_OF_CITIES][NUMBER_OF_CITIES];
```

- we can then access any of the 49 elements by using double subscripting
- `distances[2][3] = 1661;`

Declaring 2-D Array

- a 2-D array is declared with two sets of square brackets

```
const int COUNTRIES = 5;
const int MEDALS = 3;
unsigned medal_counts[COUNTRIES][MEDALS];
```

- you can initialize

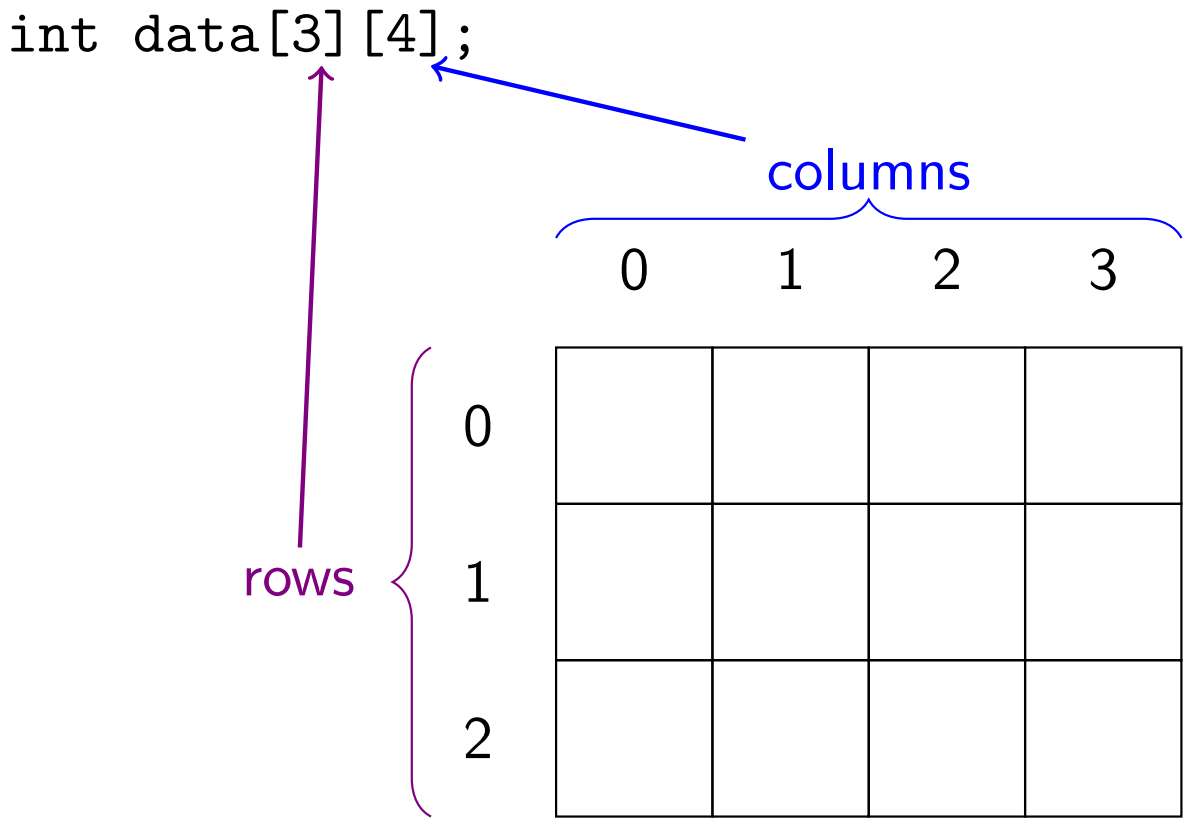
```
unsigned medal_counts[COUNTRIES][MEDALS] {{1, 0, 1},
                                          {1, 1, 3},
                                          {2, 0, 1},
                                          {4, 4, 1},
                                          {0, 5, 2}};
```

Coordinate Models

- for 2-D data, there are four competing mental images
  1. a position can be denoted row-then-column, e.g., on a page of text, line 5, word 11
  2. a position can be denoted column-then-row coordinates, e.g., on a spreadsheet, cell D14
  3. a position can be denoted by x, y coordinates
     - 3.1 y-coordinate increases up, e.g., the Cartesian plane
     - 3.2 y-coordinate increases down, e.g., pixels on a screen
- thus using i and j as variable names for the coordinates of a cell is unacceptable (even though you see this done lots)
- you must use variable names that explicitly state which mental model you are using

Mental Model

- when you use a 2-D array, you must have a clear mental picture of your model



Higher Dimensions

- you can have more than two dimensions
- but it's rare to have more than three in a real program
- sometimes in specialized software e.g. for chemistry or physics, but not much else

Declaring

- for a 1-D array, you can leave out the size if you initialize `int values[] {1, 2, 3};`
- the same is almost true of a 2-D array
- you can leave out only the last (leftmost) dimension

```
unsigned counts[][MEDALS] {{1, 0, 1},
                           {4, 5, 3},
                           {7, 1, 1},
                           {3, 7, 2},
                           {0, 9, 4},
                           {2, 0, 1}};
```

Passing 2-D Arrays

- the same thing is true of passing a 2-D array as a parameter
- you must specify all dimensions except the leftmost

```
void show_array(const int numbers[][COLUMNS],
                unsigned rows);
```

Summing All the Elements of a 2-D Array

- a pair of nested for loops is ideal for processing the elements of a 2-D array

```
const unsigned NUM_ROWS = 5;
const unsigned NUM_COLS = 7;

int total = 0;
int values[NUM_ROWS][NUM_COLS] { ... };

for (unsigned row = 0; row < NUM_ROWS; row++)
{
  for (unsigned col = 0; col < NUM_COLS; col++)
  {
    total += values[row][col];
  }
}
```

Arrays

- arrays are insanely useful
- and indispensable for programming
- and essential to understanding many crucial computer science concepts

Arrays

- arrays are insanely useful
- and indispensable for programming
- and essential to understanding many crucial computer science concepts
- BUT

Arrays

- arrays are insanely useful
- and indispensable for programming
- and essential to understanding many crucial computer science concepts
- BUT
- they are essentially never used in modern application program development
- because arrays have some huge issues

Array Problems

- array size is static
- array size must be known at compile time
- arrays do not know how big they are

Array Problems

- array size is static
- array size must be known at compile time
- arrays do not know how big they are

- what if we could have an array that could be declared using a variable instead of a const?
- one that could shrink and grow as needed, never wasting space?
- one that knew how big it was, so we didn't need to pass an additional size parameter?

Array Problems

- array size is static
- array size must be known at compile time
- arrays do not know how big they are

- what if we could have an array that could be declared using a variable instead of a const?
- one that could shrink and grow as needed, never wasting space?
- one that knew how big it was, so we didn't need to pass an additional size parameter?

- we do — it's called vector

The STL

- the Standard Template Library is a collection of data structures that real programs use
- one of the most-used is the vector, like an array on steroids
- in fact, vector has an actual array under the hood, but the programmer doesn't have to interact with it directly

The STL

- the Standard Template Library is a collection of data structures that real programs use
- one of the most-used is the vector, like an array on steroids
- in fact, vector has an actual array under the hood, but the programmer doesn't have to interact with it directly

- you will have to understand, use, and interact with arrays for this course and for computer science in general
- but for real programs you will use vectors

Vector

- you must include the vector library header
  `#include <vector>`
- declare a vector variable:
  `vector<int> values;`
  this creates a vector of zero size

Vector

- you must include the vector library header
  `#include <vector>`
- declare a vector variable:
  `vector<int> values;`
  this creates a vector of zero size

- declare a vector variable with a specific size:
  `vector<int> values(10);`
  this declares a vector with room for 10 ints

Vector

- you must include the vector library header
  `#include <vector>`
- declare a vector variable:
  `vector<int> values;`
  this creates a vector of zero size

- declare a vector variable with a specific size:
  `vector<int> values(10);`
  this declares a vector with room for 10 ints

- declare a vector variable with a size and initialized all to one value:
  `vector<double> values(10, 0.0);`

Vector

- declare and initialize a vector with an initialization list
  `vector<unsigned> values {1, 3, 5, 7, 9};`

Vector

- declare and initialize a vector with an initialization list
  `vector<unsigned> values {1, 3, 5, 7, 9};`

- start with an empty vector and one by one add five values

```
vector<string> names; // names is size zero

names.push_back("Ann"); // names has grown to size one
names.push_back("Bob"); // names has two elements
names.push_back("Cal");
names.push_back("Deb");
names.push_back("Eli");
```

Vector

- find out how many elements are currently in a vector
  ```
  size_t number_of_elements = values.size();
  ```
- size_t is a built-in unsigned type that is guaranteed to work across hardware and operating system platforms
- whenever you use a library routine to find out a size in C++, the return type will be size_t
- often it's the same as simple unsigned, but not always

Vector

- to access a vector element, do NOT use square brackets like Gaddis does
- use .at()

```
for (size_t index = 0; index < values.size(); index++)
{
  values.at(index) *= 1.1; // give everyone a 10% bonus
}
```

Vector

- to access a vector element, do NOT use square brackets like Gaddis does
- use .at()

```
for (size_t index = 0; index < values.size(); index++)
{
  values.at(index) *= 1.1; // give everyone a 10% bonus
}
```

- remove the last element from a vector
  `values.pop_back();`
- remove all values from a vector, setting its size to zero:
  `values.clear();`

Vector Parameters

- to pass a vector to a function, it is legal to pass by value, but you should never do this!
- copying a vector takes time proportional to the number of elements — could be huge
- there are also potentially bad side effects we will study later
- bottom line: always pass a vector either by reference (if you're going to change it) or by const reference (if you're not going to change it)

```
void show_values(const vector<int>& values);
void double_the_values(vector<int>& values);
```