CSC 221: Computer Programming I

Fall 2001

- arrays
  - ➢ homogeneous collection of items, accessible via an index
  - ➢ initializing/traversing/displaying arrays
  - ➢ array initialization
  - ➢ arrays as references

    out-of-bounds indexing, parameter passing, no assignment

recall dice stats example

last implementation involved a function for repeated simulations

- can call for each dice total, but each call is an independent experiment

```
int CountDice(int numRolls, int desiredTotal)
// Assumes: numRolls >= 0, 2 <= desiredTotal <= 12
// Returns: number of times desiredTotal occured out of numRolls simulated Dice rolls
{
          Die d1, d2;

          int desiredCount = 0;
          while (d1.NumRolls() < numRolls) {
              int roll1 = d1.Roll();
              int roll2 = d2.Roll();

              if (roll1 + roll2 == desiredTotal) {
                  desiredCount++;
              }
          }

          return desiredCount;
}
```

```
for (int roll = 2; roll <= 12; roll++) {
    cout << setw(4) << CountDice(NUM_ROLLS,  roll) << " were "
         << setw(2) << roll << "'s." << endl;
}
```

new dice stats

want to perform a single experiment, count all totals simultaneously

- need a single loop (as opposed to independent loops)
- need a counter for each possible dice total

```
int count2 = 0,  count3 = 0,
    count4 = 0,  count5 = 0,
    count6 = 0,  count7 = 0,
    count8 = 0,  count9 = 0,
    count10 = 0, count11 = 0,
    count12 = 0;
```

how do we display the counts?

can we roll the dice in one function, display counts in another?

```
for (int rep = 1; rep <= NUM_REPS; rep++) {
    int total = d1.Roll() + d2.Roll();
    if (total == 2) {
        count2++;
    }
    else if (total == 3) {
        count3++;
    }
    else if (total == 4) {
        count4++;
    }
    else if (total == 5) {
        count5++;
    }
    else if (total == 6) {
        count6++;
    }
    else if (total == 7) {
        count7++;
    }
    else if (total == 8) {
        count8++;
    }
    else if (total == 9) {
        count9++;
    }
    else if (total == 10) {
        count10++;
    }
    else if (total == 11) {
        count11++;
    }
    else {
        count12++;
    }
}
```

drawbacks of having 11 counters!

- declaring, initializing, & maintaining 11 counters is tedious
  *** typos are inevitable ***
- to roll dice & update correct counter, need 11 case cascading if-else
- to display counts, need 11 cout statements
- if want to modularize the code (i.e., break into independent functions), passing counters requires 11 parameters!

BIG ISSUE: each variable is independent of the others

- even though `count2` and `count3` are related & similar, nothing in the code allows for systematic treatment
- conceptually, we want to be able to refer to an arbitrary `countX` for any $2 \leq x \leq 12$

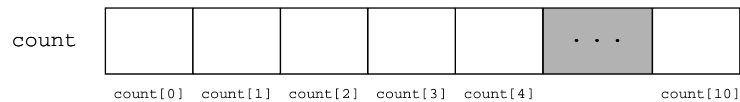
C++ arrays

in C++, an *array* allows for related values to be

- stored under a single name, accessed via an index
- traversed & operated on systematically using a loop
- passed as a single entity to and from functions

FORMAL DEFINITION: an array is a *homogeneous* collection of values, with individual values accessible via an *index*

- homogeneous → all values in the array must be of the same type
- indexable → can access by a value's position in the array
  (first value is at index 0, second value is at index 1, …)
  thus, can loop through the values in an array



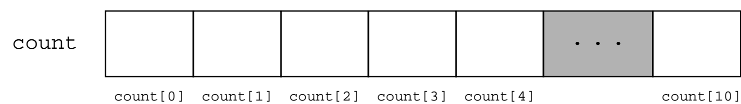
C++ arrays

to declare an array, must specify the size in brackets after the name

```
int count[11];        // declares an array of 11 ints, named count
string words[100];    // declares an array of 100 strings, named words
```

to access a particular value (a.k.a. an element) in the array, specify the array name followed by the corresponding index in brackets

```
count[0] = 0;         // initializes the first element in count to 0
cout << words[99];    // displays the last element in words
```



C++ arrays

since elements are accessed via the index (an int), can use a loop to traverse over the entire range of elements

```
// initialize all counts in array
for (i = 0; i < 11; i++) {
    count[i] = 0;
}
```

```
// display all words in array
for (j = 0; j < 100; j++) {
    cout << words[i] << endl;
}
```

QUESTION: why does the for loop test involve < and not <= ?

an array is a single entity, so can be passed as one

```
void Initialize(int nums[], int size)
// Assumes: nums contains size integer values
// Results: all elements of nums are set to 0
{
    for (int i = 0; i < size; i++) {
        nums[i] = 0;
    }
}
```

when specify an array as a parameter, don't specify size (must pass separately if needed)

arrays are automatically passed by-reference **(EXPLANATION LATER)**

arrays & dice stats

arrays provide the perfect improvement to the dice stats program

- have an array of counters, one for each dice total
  ```
  int diceCounts[11];
  ```
- for each dice roll, add to the corresponding counter in the array

  dice total 2 ➔ `diceCounts[0]++;`

  dice total 3 ➔ `diceCounts[1]++;`

  . . .

  dice total 12 ➔ `diceCounts[10]++;`
- can initialize & display counters using a for loop
  ```
  for (i = 0; i < 11; i++) {
      diceCounts[i] = 0;
  }
  ```
- since the dice counts are stored together in an array, can pass as a single parameter

dice stats w/ arrays

```
// dice.cpp
//
// Uses an array of counters to keep track
// of random dice rolls.
/////////////////////////////////////////

#include <iostream>
#include "Die.h"
using namespace std;

const int NUM_ROLLS=1000;

void Initialize(int nums[]);
void DoRolls(int count[], int numberRolls);
void DisplayCounts(int count[]);

int main()
{
    int diceCounts[11];
    Initialize(diceCounts);

    DoRolls(diceCounts, NUM_ROLLS);

    DisplayCounts(diceCounts);

    return 0 ;
}

/////////////////////////////////////////
```

```
void Initialize(int count[])
// Assumes: count contains 11 counters
// Results: initializes all values to 0
{
    for (int i = 0; i < 11; i++) {
        count[i] = 0;
    }
}

void DoRolls(int count[], int numberRolls)
// Assumes: count contains 11 counters
// Results: performs numberRolls dice simulations
//          and adds to each corr. counter
{
    Die d1(6), d2(6);

    for (int i = 0; i < numberRolls; i++) {
        int total = d1.Roll() + d2.Roll();
        count[total-2]++;
    }
}

void DisplayCounts(int count[])
// Assumes: count contains 11 counters
// Results: displays each counter value
{
    for (int i = 0; i < 11; i++) {
        cout << "# of " << (i+2) << "'s = "
             << count[i] << endl;
    }
}
```

generalized dice stats w/ arrays

can generalize to handle arbitrary dice (constant specifies # sides)

```
// dice.cpp
//
// Uses an array of counters to keep track
// of random dice rolls.
/////////////////////////////////////////

#include <iostream>
#include "Die.h"
using namespace std;

const int DIE_SIDES = 6;
const int NUM_ROLLS=1000;

void Initialize(int nums[]);
void DoRolls(int count[], int numberRolls);
void DisplayCounts(int count[]);

int main()
{
    int diceCounts[2*DIE_SIDES-1];
    Initialize(diceCounts);

    DoRolls(diceCounts, NUM_ROLLS);

    DisplayCounts(diceCounts);

    return 0 ;
}

/////////////////////////////////////////
```

```
void Initialize(int count[])
// Assumes: count contains 2*DIE_SIDES-1 counters
// Results: initializes all values to 0
{
    for (int i = 0; i < 2*DIE_SIDES-1; i++) {
        count[i] = 0;
    }
}

void DoRolls(int count[], int numberRolls)
// Assumes: count contains 2*DIE_SIDES-1 counters
// Results: performs numberRolls dice simulations
//          and adds to each corr. counter
{
    Die d1(DIE_SIDES), d2(DIE_SIDES);

    for (int i = 0; i < numberRolls; i++) {
        int total = d1.Roll() + d2.Roll();
        count[total-2]++;
    }
}

void DisplayCounts(int count[])
// Assumes: count contains 2*DIE_SIDES-1 counters
// Results: displays each counter value
{
    for (int i = 0; i < 2*DIE_SIDES-1; i++) {
        cout << "# of " << (i+2) << "'s = "
             << count[i] << endl;
    }
}
```

mismatched indices

confusing feature of this program

- dice totals range 2 to 12; counter indices range 0 to 10

➔ each roll of the dice must get mapped to the corresponding index

```
for (int i = 0; i < numberRolls; i++) {
      int total = d1.Roll() + d2.Roll();
      count[total-2]++;
}
```

➔ when displaying the counts, must also map totals and indices

```
for (int i = 0; i < 2*DIE_SIDES-1; i++) {
      cout << "# of " << (i+2) << "'s = "
               << count[i] << endl;
}
```

one solution: allocate extra spaces so that totals & indices can match

dice total 2 ➔ `diceCounts[2]++;`

. . .

dice total 12 ➔ `diceCounts[12]++;`

dice stats (yet again)

```
// dice.cpp
//
// Uses an array of counters to keep track
// of random dice rolls.
/////////////////////////////////////////

#include <iostream>
#include "Die.h"
using namespace std;

const int DIE_SIDES = 6;
const int NUM_ROLLS=1000;

void Initialize(int nums[]);
void DoRolls(int count[], int numberRolls);
void DisplayCounts(int count[]);

int main()
{
    int diceCounts[2*DIE_SIDES+1];
    Initialize(diceCounts);

    DoRolls(diceCounts, NUM_ROLLS);

    DisplayCounts(diceCounts);

    return 0 ;}

/////////////////////////////////////////
```

```
void Initialize(int count[])
// Assumes: count contains 2*DIE_SIDES+1 counters
// Results: initializes all values to 0
{
    for (int i = 2; i <= 2*DIE_SIDES; i++) {
        count[i] = 0;
    }
}

void DoRolls(int count[], int numberRolls)
// Assumes: count is an array of 2*DIE_SIDES+1
    counters
// Results: performs numberRolls dice simulations
//          and adds to each corr. counter
{
    Die d1(DIE_SIDES), d2(DIE_SIDES);

    for (int i = 0; i < numberRolls; i++) {
          int total = d1.Roll() + d2.Roll();
          count[total]++;
    }
}

void DisplayCounts(int count[])
// Assumes: count contains 2*DIE_SIDES+1 counters
// Results: displays each counter (ignores 0 & 1)
{
    for (int i = 2; i <= 2*DIE_SIDES; i++) {
          cout << "# of " << i << "'s = "
                      << count[i] << endl;
    }
}
```

array initialization

if you know exactly what values you want to store in an array, there is a shortcut notation for declaring and initializing

```
int diceCount[] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

string grades[] = {"A", "B+", "B", "C+", "C", "D", "F"};
```

- array elements are specified in curly-braces, separated by commas
- no need to specify size (compiler can count the number of values)

in the first example: saves a loop for initializing all 12 counters
in the second example: saves 7 different assignments

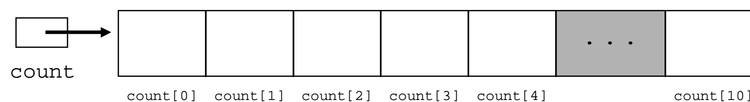
reconsider dice stats:

- if the number of die sides is set, could use this shorthand to initialize array
- if die sides is to be changeable, must explicitly declare & init with a loop

arrays as references

interesting fact about arrays

- because arrays can store a lot of data, it is more efficient to refer to the array as a pointer (i.e., an address) to the actual memory cells



- when you index an array, the index specifies an offset amount from the initial pointer
  `count[0]` → go to where count points (first memory cell in the array)
  `count[1]` → go to where count points plus 1 `int` farther down
  `count[2]` → go to where count points plus 2 `int`'s farther down
  ...

  this explains why arrays are homogeneous – must know how big to step

  it is possible to specify an index that is out-of-bounds of the array
  `count[11]` → go to where count points plus 11 `int`'s farther down
  `count[-1]` → go to where count points minus 1 `int` farther back

OOB behavior

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

int main()
{
  int nums1[10], nums2[5];

  for (int i = 0; i < 10; i++) {
    nums1[i] = i;
  }

  for (int j = 0; j < 5; j++) {
    nums2[j] = 100+j;
  }

  for (int k = 0; k < 10; k++) {
    cout << k << ": " <<nums2[k] << endl;
  }

  return 0;
}
```

using Visual C++, this program produces the following:

```
0: 100
1: 101
2: 102
3: 103
4: 104
5: 0
6: 1
7: 2
8: 3
9: 4
```

WHY?

```
  cout << nums1[1000] << endl;
```

this attempted access will crash the program. WHY?

if the memory cell referenced by the OOB index belongs to the program, then OK (?)
if the cell is outside the program's allotted memory, then CRASH!

arrays as parameters

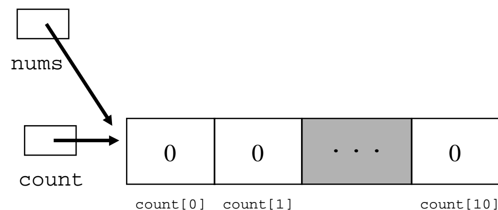
the fact that arrays are treated as references to the memory cells has implications for parameter passing

```
function foo(int nums[])
{
    nums[0] = 1234;
}

. . .

int count[10];
for (int i = 0; i < 10; i++) {
    count[i] = 0;
}

foo(count);
```



- when you pass an array as a parameter, you are really passing a pointer
- even though the parameter `nums` is a copy of `count`, it still provides access to the memory cells in the array, so can make changes to the data elements

➔ when you pass an array by-value, it looks like by-reference!

array assignments

another result of the fact that arrays are really pointers to cells, you are not allowed to copy arrays as single entities

```
int count[] = { 1, 2, 3, 4, 5};

int copy[5];
copy = count;          // ILLEGAL: CAUSES COMPILE-TIME ERROR
```

what would be the effect if this were allowed?

instead, you must copy element-by-element

```
int copy[5];
for (int i = 0; i < 5; i++) {
    copy[i] = count[i];
}
```

silly array example

many tasks involve reading in and manipulating an arbitrary number of values

silly example: suppose we want to read in a series of words (terminated by "DONE") and display them in reverse

- must read in words, store in an array, and keep track of how many stored
- once read, can display by traversing in reverse order

pseudo-code:

```
CREATE ARRAY;
SET ELEMENT COUNT TO 0;

READ FIRST WORD;
while (WORD IS NOT "DONE") {
    ADD WORD TO AN ARRAY;
    INCREMENT ELEMENT COUNT;

    READ NEXT WORD;
}

for (i RANGING BACKWARDS FROM COUNT) {
    DISPLAY WORD AT INDEX i;
}
```

reverse array

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

const int MAX_SIZE = 100;

int main()
{
  string words[MAX_SIZE];
  int numWords = 0;

  string input;
  cin >> input;
  while (input != "DONE") {
    words[numWords] = input;
    numWords++;

    cin >> input;
  }

  for (int i = numWords-1; i >= 0; i--) {
    cout << words[i] << endl;
  }

  return 0;
}
```

NOTE: the size of the array must be specified when it is declared (i.e., at compile-time)

since you don't know how many words there will be, must pick a size big enough to accommodate any reasonable input sequence

as a result, much of this space may be wasted on any given execution

initial array summary

arrays provide capability for storing related values in a single entity

- can access individual values using an index
- can traverse through the indices, systematically access all values in the array
- can pass the entire array as a single parameter

a short-hand notation exists for creating and initializing arrays

- useful when the size is small, know initial values ahead of time

for efficiency reasons, arrays are represented as references (pointers) to the collection of memory cells

- accessing an index involves computing the offset from the initial pointer
  →out-of-bounds accesses are usually not caught
- even when passed as value parameter, behavior is like reference parameter
- arrays cannot be assigned as a whole