# C Arrays

## CS 2060 Week 5

Prof. Jonathan Ventura

1 Arrays
- Arrays
- Initializing arrays

2 Examples of array usage

3 Passing arrays to functions

4 2D arrays
- 2D arrays

5 Strings
- Using character arrays to store and manipulate strings

6 Searching arrays
- Searching arrays

7 Next Time

# Arrays

- An array is a group of *contiguous* memory locations that all have the *same type*.
- To refer to a particular location or element in the array, we specify the array's name and the **position number** (or **array index**) of that particular element.

# Arrays

| Array element | Value |
|---|---|
| c[0] | 0 |
| c[1] | 1 |
| c[2] | 1 |
| c[3] | 2 |
| c[4] | 3 |
| c[5] | 5 |
| c[6] | 8 |
| c[7] | 13 |
| c[8] | 21 |
| c[9] | 34 |

Table: Array example: an array of 10 integers named c.

# Accessing arrays

- Array indices begin at zero.
- Square brackets are used to access an array element:

```
c[0] = 0;
c[1] = 1;
c[n] = c[n-2] + c[n-1];
printf("%d ",c[n]);
```

# Defining arrays

- An array is defined like a normal variable, but with square brackets indicating the array size.

```
int c[10];
```

- The above statement defines an array of ten integers, with indices 0-9.
- Multiple arrays can be defined in a single statement:

```
char a[100], b[2];
```

# Array size

- The array size is fixed and cannot be changed after the definition.
- Typically, the array size must be a constant expression, not a variable:

```
int a = 10;
int b[a]; // not allowed
```

- However, this is now defined in the C99 standard, so modern compilers may allow it.

# Fibonacci example

```c
#include <stdio.h>

int main()
{
    int fibonacci[10];

    fibonacci[0] = 0;
    fibonacci[1] = 1;
    for ( int i = 2; i < 10; i++ )
    {
        fibonacci[i] = fibonacci[i-2] + fibonacci[i-1];
    }

    for ( int i = 0; i < 10; i++ )
    {
        printf( "%d ", fibonacci[i] );
    }
}
```

# Initializing arrays with a loop

- The array elements are uninitialized by default.
- The array can be initialized in several ways.

- With a loop:

```
for ( int i = 0; i < 10; i++ ) a[i] = 0;
```

# Initializing arrays with an initializer list

- An initializer list can be used to explicitly initialize the array when it is defined.

```
int a[5] = { 0, 1, 2, 3, 4 };
```

- The initializer list can *only* be used when the array is first defined.

# Initializing arrays with an initializer list

- If the initializer list has fewer elements than the array size, the remaining elements will be initialized to zero.

```
int a[5] = { 10 };
```

is equivalent to:

```
int a[5] = { 10, 0, 0, 0, 0 };
```

# Initializing arrays with an initializer list

- If the array size is *omitted*, then the size will be inferred from the initializer list.

```
int a[] = { 0, 1, 2, 3, 4, 5 };
```

is equivalent to:

```
int a[6] = { 0, 1, 2, 3, 4, 5 };
```

# Specifying an array's size with a symbolic constant

- In some cases it is convenient to put an array's size in a **symbolic constant** which is available anywhere in the code file.

```
#define BUF_SIZE 1024

int buffer[BUF_SIZE];
```

- `#define` is a **preprocessor directive**; the preprocessor will replace every instance of `BUF_SIZE` with 1024.
- The `#define` directive must occur outside any function (usually at the top of the file after the `#include` statements).
- Again, Google and Apple would prefer `kBufferSize` instead of `BUF_SIZE`.

# Specifying an array's size with a symbolic constant

```c
#include <stdio.h>
#define kBufferSize 1024

int main(void)
{
    int buffer[kBufferSize] = { 0 };

    int numValuesRead = 0;
    int value;
    while ( scanf( "%d", &value ) == 1 && numValuesRead < kBufferSize )
    {
        buffer[numValuesRead++] = value;
    }

    printf("Read %d values.\n", numValuesRead );

    int sum = 0;
    for ( int i = 0; i < numValuesRead; i++ )
    {
        sum += buffer[i];
    }

    int average = sum/numValuesRead;

    printf("Average value: %d\n", average );
}
```

# Using an array to count survey responses

```c
#include <stdio.h>

int main()
{
    // survey has responses on scale from 1-7
    unsigned int responseCount[8] = { 0 };

    int response;
    while ( scanf( "%d", &response ) == 1 )
    {
        responseCount[response]++;
    }

    printf( "%6s%17s\n", "Rating", "Frequency" );

    for ( int rating = 1; rating <= 7; rating++ )
    {
        printf( "%6d%17d\n", rating, responseCount[rating] );
    }
}
```

# Array out-of-bounds checking

- C does not provide any bounds checking on the array index.

```
int a[10] =  0 ;
a[10] = 10; // not a compiler error
a[-1] = 10; // not a compiler error
```

- Trying to read or write a value outside of the array will probably crash the program – but not always!
- This can lead to hard-to-find bugs.

# Array out-of-bounds checking

- In the previous code example, we do not check the range of the input.

```
while ( scanf( "%d", &response ) == 1 )
    responseCount[response]++;
```

- `response` could be below 1 or above 7.
- We should explicitly check the range in the code.

# Using an array to count survey responses

```c
#include <stdio.h>

int main()
{
    // survey has responses on scale from 1-7
    unsigned int responseCount[8] = { 0 };

    int response;
    while ( scanf( "%d", &response ) == 1 )
    {
        if ( response >= 1 && response <= 7 ) responseCount[response]++;
    }

    printf( "%6s%17s\n", "Rating", "Frequency" );

    for ( int rating = 1; rating <= 7; rating++ )
    {
        printf( "%6d%17d\n", rating, responseCount[rating] );
    }
}
```

# Printing array elements with a histogram

```c
#include <stdio.h>

int main()
{
    // survey has responses on scale from 1-7
    unsigned int responseCount[8] = { 0 };

    int response;
    while ( scanf( "%d", &response ) == 1 )
    {
        if ( response >= 1 && response <= 7 ) responseCount[response]++;
    }

    printf( "%6s%17s\n", "Rating", "Histogram" );

    for ( int rating = 1; rating <= 7; rating++ )
    {
        printf( "%6d         ", rating );
        for ( int i = 1; i <= responseCount[rating]; i++ )
        {
            printf("%c", '*');
        }

        puts("");
    }
}
```

# Counting die rolls

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define kNumTrials 60000000

int main()
{
    unsigned int faceCount[6] = { 0 };

    for ( unsigned int i = 0; i < kNumTrials; i++ )
    {
        int face = rand() % 6;
        faceCount[face]++;
    }

    printf( "%6s%17s\n", "Face", "Frequency" );

    for ( int face = 0; face < 6; face++ )
    {
        printf( "%6d%17u\n", face+1, faceCount[face] );
    }
}
```

# Passing arrays to functions

- Functions can receive arrays as arguments.

```
int sumArray( int array[] ) { a[0] += 1; }
```

- In the function prototype, you write the array name followed by square brackets.
  - The array size is omitted.

# Passing arrays to functions

```c
#define kArraySize 5

void printArray( int b[] );
void modifyArray( int b[] );
void modifyElement( int e );

int main() {
    int a[kArraySize] = { 0 };

    for ( int i = 0; i < kArraySize; i++ ) a[i] = i;

    printArray( a );
    modifyArray( a ); printArray( a );
    modifyElement( a[3] ); printArray( a );
}

void printArray( int b[] ) {
    for ( int i = 0; i < kArraySize; i++ ) printf( "%3d", b[i] );
}

void modifyArray( int b[] ) {
    for ( int i = 0; i < kArraySize; i++ ) b[i] *= 2;
}

void modifyElement( int e ) {
    e *= 2;
}
```

# const arrays

- Using the const modifier we can prevent modifications to an array.

```
int myfn( const int a[] ) {
    int b = a[0]; // allowed
    a[0] += 1; // not allowed
}
```

- The const modifier is useful for:
  - Specifying input arguments
  - Creating arrays of fixed constants
- Any variable (not just arrays) can be declared const.

# const array example

```c
#define kArraySize 10

const int gFibonacci10[] = { 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 };

void copyArray( int output[], const int input[] );
void squareArray( int output[], const int input[] );

int main() {
    int fib[kArraySize] = { 0 };
    int fib_squared[kArraySize] = { 0 };

    copyArray( fib, gFibonacci10 );
    squareArray( fib_squared, fib );
}

void copyArray( int output[], const int input[] ) {
    for ( int i = 0; i < kArraySize; i++ ) output[i] = input[i];
}

void squareArray( int output[], const int input[] ) {
    for ( int i = 0; i < kArraySize; i++ ) output[i] = input[i]*input[i];
}
```

# const array example

```
#define kArraySize 10

const int gFibonacci10[] = { 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 };

void copyArray( int output[], const int input[] );
void squareArray( int output[], const int input[] );

int main() {
    int fib[kArraySize] = { 0 };

    copyArray( fib, gFibonacci10 );
    squareArray( fib, fib );
}

void copyArray( int output[], const int input[] ) {
    for ( int i = 0; i < kArraySize; i++ ) output[i] = input[i];
}

void squareArray( int output[], const int input[] ) {
    for ( int i = 0; i < kArraySize; i++ ) output[i] = input[i]*input[i];
}
```

# Multidimensional arrays

- With **multidimensional arrays** we use multiple indices to refer to array elements.

```
c[0][0] = 0;
```

- Two-dimensional arrays are common:
  - Tables
  - Matrices
- The first index indicates the *row* and the second index indicates the *column*.

# Two-dimensional array example

| | Column 0 | Column 1 | Column 2 | Column 3 |
|---|---|---|---|---|
| Row 0 | a[0][0] | a[0][1] | a[0][2] | a[0][3] |
| Row 1 | a[1][0] | a[1][1] | a[1][2] | a[1][3] |
| Row 2 | a[2][0] | a[2][1] | a[2][2] | a[2][3] |
| Row 3 | a[3][0] | a[3][1] | a[3][2] | a[3][3] |

Table: Two-dimensional array elements in a 4x4 array

# Row major order

- Like linear (1D) arrays, multidimensional arrays are contiguous blocks of memory.

- C stores multidimensional arrays in **row major order**.
- This means that the first row of data is stored before the second, and so on.

# Initializing 2D arrays with a loop

- A 2D array can be initialized in several ways.
- With a nested loop:

```
int a[5][5];
for ( int row = 0; row < 5; row++ ) {
    for ( int col = 0; col < 5; col++ ) {
        a[row][col] = row+col;
    }
}
```

# Initializing 2D arrays with an initializer list

- An initializer list can be used to explicitly initialize the 2D array when it is defined.

```
int a[5][5] = {
    { 0, 1, 2, 3, 4 },
    { 1, 2, 3, 4, 5 },
    { 2, 3, 4, 5, 6 },
    { 3, 4, 5, 6, 7 },
    { 4, 5, 6, 7, 8 }
};
```

- The initializer list can *only* be used when the array is first defined.

# Initializing arrays with an initializer list

- If *a row* in the initializer list has fewer elements than the array size, the remaining elements *in that row* will be initialized to zero.

```
int a[2][3] = { { 10 }, { 10 } };
```

is equivalent to:

```
int a[2][3] = { { 10, 0, 0 }, { 10, 0, 0 } };
```

# Initializing arrays with an initializer list

- Only the *number of rows* can be inferred by omitting the size:

```
int a[][2] = { { 1, 0 }, { 0, 1 } };
```

- The 2nd and following dimensions must be explicitly given.

# Passing multidimensional arrays to functions

- When passing a multidimensional array to a function, the size of the first dimension is omitted, but the remaining sizes must be given.

```
void myfn( int a[][2] ) { return a[0][0]; }
```

- Why?

# Passing multidimensional arrays to functions

- The second (and following) dimension sizes determine the indexing function for the multidimensional array.

- To illustrate this, we can replace a 2D array with a 1D array in the following way:

```
int a[3][10];
a[0][2] = 10;
a[1][2] = 10;

int b[3*10];
b[0*10 + 2] = 10;
b[1*10 + 2] = 10;
```

# Arrays of other types

- We can define an array on any data type, even an `enum`.

```
enum Color { RED, BLUE, GREEN };
enum Color image[480][640];

bool tests[100][100];
float matrix[3][3];
```

# 2D array example

```c
#include <stdio.h>
int sumRow( int magic[][3], int row );
int sumCol( int magic[][3], int col );
int main() {
    int magic[3][3];

    for ( int row = 0; row < 3; row++ ) {
        for ( int col = 0; col < 3; col++ ) {
            scanf("%d",&magic[i][j]);
        }
    }

    int row_sums[3], column_sums[3], diagonal_sums[2];

    for ( int row = 0; row < 3; row++ ) row_sums[row] = sumRow( magic, row );
    for ( int col = 0; col < 3; col++ ) col_sums[col] = sumColumn( magic, col )
    diagonal_sums[0] = magic[0][0] + magic[1][1] + magic[2][2];
    diagonal_sums[1] = magic[0][2] + magic[1][1] + magic[2][0];
}
```

# 2D array example

```
int sumRow( int magic[][3], int row )
{
    int sum = 0;
    for ( int col = 0; col < 3; col++ ) sum += magic[row][col];
    return sum;
}

int sumColumn( int magic[][3], int col )
{
    int sum = 0;
    for ( int row = 0; row < 3; row++ ) sum += magic[row][col];
    return sum;
}
```

# Storing strings in character arrays

- We can use an array to store a string:

```
char string1[] = "first";
```

# Storing strings in character arrays

- We can use an array to store a string:

```
char string1[] = "first";
```

- In C, strings end with a special *string-termination character* known as a **null character**.
  - C strings are called **null-terminated strings**.

# Storing strings in character arrays

- We can use an array to store a string:

```
char string1[] = "first";
```

- In C, strings end with a special *string-termination character* known as a **null character**.
  - C strings are called **null-terminated strings**.
- The above is equivalent to:

```
char string1[] = {'f','i','r','s','t','\0'};
```

# Storing strings in character arrays

- Remember that:
  - Single quotes are for single characters:
    `char letter = 'a';`
  - Double quotes are for null-terminated strings:
    `char word[] = "hello";`

# Storing strings in character arrays

- Remember that:
  - Single quotes are for single characters:
    `char letter = 'a';`
  - Double quotes are for null-terminated strings:
    `char word[] = "hello";`
- These statements will not work:
  - `char letter = "a";`
  - `char word[] = 'hello';`

# Accessing characters in a string

- The individual characters in a string are accessed with normal array indices:

```
char word[] = "hello";
char letter = word[0];
```

# Accessing characters in a string

- The individual characters in a string are accessed with normal array indices:

```
char word[] = "hello";
char letter = word[0];
```

- How long is the array `word`?

# Accessing characters in a string

- The individual characters in a string are accessed with normal array indices:

```
char word[] = "hello";
char letter = word[0];
```

- How long is the array `word`?
- Six characters – five letters plus the null-terminator.

# Printing strings

- We have already been using strings to output text and values with `printf`:

```
char word[] = "hello";
printf("%s\n", word);
```

# Printing strings

- We have already been using strings to output text and values with `printf`:

```
char word[] = "hello";
printf("%s\n", word);
```

- The format specifier itself is a string.

# Printing strings

- We have already been using strings to output text and values with `printf`:

```
char word[] = "hello";
printf("%s\n", word);
```

- The format specifier itself is a string.
- Use `%s` to specify a string in the format specifier.

# Reading strings from input

- To read a string from input, we can use %s as well.

```
char word[20];
scanf("%s", word); // OR:
scanf("%19s", word);
```

- %19s means, read up to 19 characters (stopping at whitespace).

# Reading strings from input

- To read a string from input, we can use %s as well.

```
char word[20];
scanf("%s", word); // OR:
scanf("%19s", word);
```

- %19s means, read up to 19 characters (stopping at whitespace).
- Why 19 characters? Why not 20?

# Reading strings from input

- To read a string from input, we can use %s as well.

```
char word[20];
scanf("%s", word); // OR:
scanf("%19s", word);
```

- %19s means, read up to 19 characters (stopping at whitespace).
- Why 19 characters? Why not 20?
- We need to leave one element for the null character at the end of the string.

# Reading strings from input

- To read a string from input, we can use %s as well.

```
char word[20];
scanf("%s", word);
```

- Why not `scanf("%s", &word);` ?

# Reading strings from input

- To read a string from input, we can use %s as well.

```
char word[20];
scanf("%s", word);
```

- Why not `scanf("%s", &word);` ?
- We don't use the & when passing an array to a function.

# String example

```c
#include <stdio.h>
int main()
{
    char word[20];
    scanf("%19s",word);

    int a_index = -1;

    // what is the error here?
    for ( int i = 0; i < 20; i++ ) {
        if ( word[i] == 'a' ) {
            a_index = i;
            break;
        }
    }

    if ( a_index == -1 ) printf("no 'a' found.\n");
    else printf("first 'a' found at index %d\n",a_index);
}
```

# String example

```c
#include <stdio.h>
int main()
{
    char word[20];
    scanf("%19s",word);

    int a_index = -1;

    for ( int i = 0; i < 20; i++ ) {
        if ( word[i] == '\0' ) break;
        if ( word[i] == 'a' ) {
            a_index = i;
            break;
        }
    }

    if ( a_index == -1 ) printf("no 'a' found.\n");
    else printf("first 'a' found at index %d\n",a_index);
}
```

# Searching arrays

- Searching an array for particular values is a common operation.
- We may want to know:
  - If a particular value occurs in the array.
  - Where the first (or last) occurence of the value is.
  - How many times the value occurs.
- The simplest approach is to use **linear search**, as demonstrated in the previous code example.

# Linear search

```
// finds the first occurence of key in array
// returns -1 if key is not found
int linearSearch( const int array[], int key, int size )
{
    for ( int n = 0; n < size; n++ ) {
        if ( array[n] == key ) return n; // return location of key
    }
    return -1; // key not found
}
```

# Linear search

```
// counts the number of occurences of key in array
int countFrequency( const int array[], int key, int size )
{
    int count = 0;
    for ( int n = 0; n < size; n++ ) {
        if ( array[n] == key ) count++;
    }
    return count;
}
```

# Next Time

- Homework 6 due Wednesday, March 2, 11:55 PM.
- Pointers