Computer Science I

Arrays

Dr. Chris Bourke
cbourke@cse.unl.edu

Outline

Part I: Introduction

Arrays

- Rarely do we deal with only one piece of data
- Usually more than one number, string, object, etc. must be stored and processed
- *Collections* of data can be stored in *arrays*
- An "array" is an *ordered series or arrangement*

Arrays

In code:

- Arrays are collections of ordered data stored *contiguously* in memory
- *ordered* is not the same as *sorted*
- Have a single identifier (name)
- Size is *fixed* when created
- You access individual elements in an array with an *index*
- Arrays are 0-indexed: first element is at index 0, the second at index 1, etc.
- An array of size n has the last element at index $n-1$
- Indexing is usually done with the square brackets `[]`

Example

index	0	1	2	3	4	5	6	7	8
contents	48	9	17	5	29	72	42	101	32

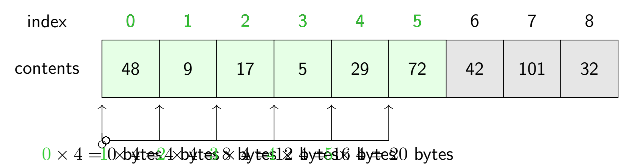
Indexing

- Suppose `arr` is an integer (4 bytes each) array
- `arr` is actually a *memory address*
- i-th element is at `arr[i]`
- Indexing automatically computes a *memory offset*
- i-the element is

$$i \times 4$$

bytes away from the beginning of the array

Memory Offsets



Part II: Using Arrays

Using Arrays

- Static arrays are allocated on the program stack
- Declaration specifies size

```c
int arr[10];
double numbers[20];
```

- Once declared, indexing can be used to access values

```c
arr[0] = 42;
arr[1] = 12;
arr[2] = arr[0] + 20;
arr[9] = 3.75; //truncation

printf("a[0] = %d\n", a[0]);
```

Alternative Syntax

Declaration/initialization

- You can declare and initialize an array at the same time

```c
int primes[] = { 2, 3, 5, 7, 11, 13, 17 };
```

- Size specification is *optional*
- Example of *code elision*
- Problem: You still need to keep track of the *size* of the array

Variable Length Arrays

- C99+ allows you to declare an array with a variable size

```c
int n = 10;
int arr[n];
```

- Just because you *can* (or more accurately *might*) be able to do this, doesn't mean you should
- Avoid in general

Pitfalls
Uninitialized Arrays

- Like regular variables, there is *no default value*
- `arr[3]` was not set, its value could be anything
- Never make assumptions about uninitialized variables
- Always initialize yourself

Pitfalls
Out-of-bounds indexing

- Accessing invalid indices is *undefined behavior*

```
int arr[10];
...
arr[10] = 42;
arr[-1] = 21;
```

- May lead to:
  - A segmentation fault, bus error
  - Corrupted memory
  - Incorrect results
- *Your* responsibility to do *bookkeeping*

Pitfalls
Book Keeping

- You must always keep track of the size of an array
- In general there is no way to determine the size of an array
- Only in very limited situations (static arrays)
- Arrays should be accompanied by an integer variable to keep track of its size
- Idiomatic loops over arrays
- Demonstration

Pitfalls
Book Keeping

```
int n = 10;
int primes[] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };
int sum = 0;

for(int i=0; i<n; i++) {
  sum += primes[i];
}
```

Part III: Dynamic Arrays

Static Arrays are Insufficient

- Static arrays are allocated on the program stack
- Inside a stack frame in which it is declared
- Stack space is *limited*
- 8MB (large) to 64k or even 8k (embedded systems)
- Demonstration

Static Arrays are Insufficient

- Stack is small, inappropriate to hold even "moderately' sized arrays
- Other disadvantages: static arrays cannot be returned from functions
- Best to not use static arrays at all
- Don't abuse the stack space, it is small and defenseless
- Better solution: use *dynamic arrays*

Dynamic Memory & Arrays

- Dynamic memory is allocated in a program's *heap*
- Stack: highly organized, efficient, but small/limited
- Heap: Less organized, less efficient, but much larger
- Dynamically allocate memory on the heap using `malloc()` (**M**emory **Alloc**ation)

malloc

- Located in the standard library `stdlib.h`
- Takes one argument: the number of bytes you want to allocate
- Use `sizeof()` to determine how many bytes each type of variable takes
- Returns a generic *void pointer*: `void *`
- A void pointer points to a generic memory location that can be *cast* to any type you want
- Returns `NULL` if unsuccessful
- Demonstration

Part IV: Memory Management

Overview

- Memory on the stack is "cleaned up" when stack frames are removed
- Memory on the heap is *not* automatically cleaned up when it is no longer needed
- It is *your* responsibility to "clean up" dynamically allocated memory when you no longer need it
- Failure to do so or failure to do so correctly can lead to:
  - Memory Leaks
  - Reduced performance
  - Illegal memory access/segmentation faults

Freeing Memory

- To clean up memory you "free" it
- Standard library function:
  `void free(void *)`
- Takes a single argument: a pointer to dynamically allocated memory
- Demonstration

Demonstration

```c
#include<stdlib.h>
#include<stdio.h>
int main(int argc, char **argv) {

  int n = 100;
  int *arr = (int *) malloc(n * sizeof(int));

  //process the array
  for(int i=0; i<n; i++) {
    arr[i] = (i+1);
  }

  free(arr);

  return 0;
}
```

Pitfall

Dangling Pointers

- Basic usage is easy, though there are many pitfalls
- Once freed, dynamic memory cannot/should not be accessed
- Best to *reset* the pointer to `NULL`
- Otherwise, it is a *dangling pointer*
- Demonstration

Pitfall

Dangling Pointers

```c
int *arr = (int *) malloc(n * sizeof(int));
//...
free(arr);

//arr still points to a memory location, but is no longer valid
printf("arr points to %p\n", arr);
//accessing is undefined behavior:
arr[0] = 42;

//best to reset to NULL;
free(arr);
arr = NULL;
```

Pitfall

Memory Ownership

- Different sections of code may “own” memory and be responsible for its management
- Stack frames are “owned” by the program/function: the program is responsible for clean up
- Ownership may be transferred: `malloc` transfers ownership to the calling function
- Ownership is a design issue/decision
- In general: only `free` memory if you own it
- Don't `free` memory before you are done with it
- Don't `free` freed memory

Pitfall

Memory Leaks

- Failure to properly clean up memory can lead to *memory leaks*
- A program holds on to memory it doesn't use
- Or: references are lost to memory that cannot be freed
- Program takes more and more resources
- Performance degrades, taking the entire system down with it
- Demonstration

Part V: Arrays & Functions

Using Arrays With Functions

- Goal: use arrays with functions
- Pass arrays to functions
- Return arrays from functions
- Recall: you always need to do your own *bookkeeping* with arrays
- Anytime you pass an array, you also need to pass its *size*
- Anytime you return an array, you need a way to implicitly determine its size

Demonstration

Write a function that takes an array of integers and returns the sum of its values.

Demonstration

```c
/**
 * This function takes an integer array (of size n) and
 * returns the sum of its elements.  It returns 0 if the
 * array is NULL.
 */
int sum(int *arr, int n) {

  if(arr == NULL) {
    return 0;
  }
  int total = 0;
  for(int i=0; i<n; i++) {
    total += arr[i];
  }
  return total;
}
```

The `const` Keyword

- Arrays are *always* passed by reference in C
- It is possible to make changes to their contents
- We don't always want this
- We can add the keyword `const` to prevent changes to the array
- The compiler checks for changes and generates an error if this "promise" is violated
- Design issue, not a full guarantee
- Demonstration

Returning Arrays

- Functions can create and return arrays
- You *cannot* return static arrays
- Only dynamic arrays can be returned
- Demonstration

Returning Arrays

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

int * foo() {
  int b[3];
  b[0] = 10;
  b[1] = 20;
  b[2] = 30;
  return b;
}

int main(int argc, char **argv) {
  int *a = foo();
  for(int i=0; i<3; i++) {
    printf("a[%d] = %d\n", i, a[i]);
  }
  return 0;
}
```

Exercises

- Write a function that takes an integer n and returns an array of n integers, all initialized to 1
- Write a function that takes an integer array and returns a new *copy* of the array with all instances of zero removed

Part VI: Multidimensional Arrays

Multidimensional Arrays

- Up to now: 1-dimensional arrays
- You can have arrays with more than one dimension
- 2-D arrays:
  - Rows & columns
  - Tabular data
  - Matrices
- 3-D arrays:
  - Rows, columns, & "lanes"
  - 3-dimensional data
- 4+ dimensional arrays:
  - Rethink what you're doing
- Focus on 2-D arrays

Multidimensional Arrays

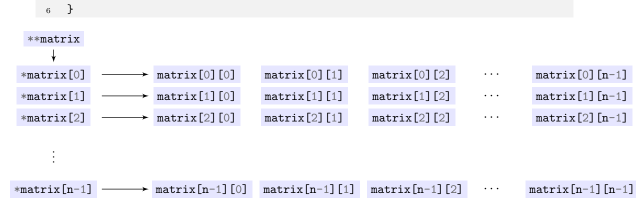
- A pointer, `int *arr` points to a 1-dimensional array
- A "double" pointer, `int **mat` points to a "2-dimensional array"
- Technically points to an array of pointers
- Each pointer in the array points to an array of integers
- Demonstration

Multidimensional Arrays

```
int n = 10;
int **matrix = NULL;
matrix = (int **) malloc(n * sizeof(int*));
for(int i=0; i<n; i++) {
  matrix[i] = (int *) malloc(n * sizeof(int));
}
```

Multidimensional Arrays

```
int n = 10;
int **matrix = NULL;
matrix = (int **) malloc(n * sizeof(int*));
for(int i=0; i<n; i++) {
  matrix[i] = (int *) malloc(n * sizeof(int));
}
```



Usage

- Once created, you can access elements using *two* indices
- Usually: row-column interpretation
- `matrix[i][j]` accesses the i-th row and j-th column
- Use two nested for-loops to iterate over each row/column

```
for(int i=0; i<n; i++) {
  for(int j=0; j<n; j++) {
    matrix[i][j] = (2*i+3*j);
  }
}
printf("last row/column value = %d\n", matrix[n-1][n-1]);
```

Demo

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

int main(int argc, char **argv) {

  int n = 3;
  int **matrix = NULL;
  matrix = (int **) malloc(n * sizeof(int*));
  for(int i=0; i<n; i++) {
    matrix[i] = (int *) malloc(n * sizeof(int));
  }

  int value = 1;
  for(int i=0; i<n; i++) {
    for(int j=0; j<n; j++) {
      matrix[i][j] = value;
      value++;
    }
```

Clean Up

- Must do proper cleanup when freeing 2-D arrays
- Cannot simply free the matrix: `free(matrix)`
- Results in a memory leak
- You must free each row before you free the array of pointers

```
for(int i=0; i<n; i++) {
  free(matrix[i]);
}
free(matrix);
```

Alternative: Contiguous Allocation

```
#include<stdlib.h>

int main(int argc, char **argv) {

  int n = 5, m = 3;
  int **arr = (int **)malloc(sizeof(int *) * n);
  arr[0] = (int *)malloc(sizeof(int) * (n * m));

  for(int i=1; i<n; i++) {
    arr[i] = (*arr + (m * i));
  }

  int value = 1;
  for(int i=0; i<n; i++) {
    for(int j=0; j<m; j++) {
      arr[i][j] = value;
      value++;
    }
```

Part VII: Shallow vs. Deep Copies

Shallow Copy

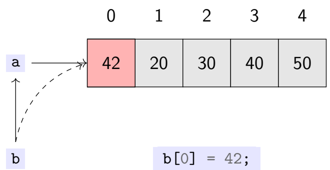
Consider the following piece of code, what does it print?

```
//create an array containing {10, 20, 30, 40, 50}
int n = 5;
int *a = (int *) malloc(n * sizeof(int));
for(int i=0; i<n; i++) {
  a[i] = (i+1)*10;
}

//let's make a "copy"
int *b = a;
b[0] = 42;

//what is in a[0]?
printf("a[0] = %d\n", a[0]);
```

Shallow Copy



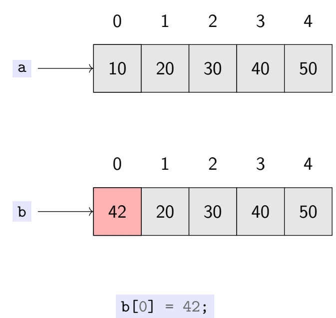
Shallow Copy

- A shared reference is a *shallow copy*
- Multiple pointers refer to the same memory location/array
- Changes to one reference affect the other
- Not typically what we want with a "copy"

Deep Copy

- In contrast: a *deep copy* is when we have two *separate* arrays with the same *contents*
- Two *different* memory locations
- Changes to one do not affect the other
- Example
- Demo: write a deep copy function

Deep Copy



Deep Copy

```c
/**
 * This function creates a deep copy of the
 * given array.
 */
int * deepCopy(const int *a, int n) {
  if(a == NULL || n < 0) {
    return NULL;
  }
  int *copy = (int *) malloc(n * sizeof(int));
  for(int i=0; i<n; i++) {
    copy[i] = a[i];
  }
  return copy;
}
```