Arrays & Dynamic Memory

Lecture Notes

Overview

- Rarely do we deal with single pieces of data; instead we have *collections* of pieces of similar data.
- Collections of data of the same type can be collected into *arrays*
- Each array has a single identifier, individual pieces of data are *elements*
- An array has a single variable identifier; individual elements are referred to by their *index*

1. Static Arrays
    a. Declaration
        i. Examples:
        ```
        int arr[10];
        double x[50];
        ```
        ii. Style debate: place square brackets on the type or the identifier:
        `int[] arr;` vs `int arr[]`
        iii. Declaration/assignment syntax:
        ```
        int a[] = { 2, 3, 5, 7, 11 };
        ```
    b. Usage
        i. Indexing: elements indexed 0 thru (size - 1)
        ii. Assignment
        iii. Accessing
        iv. Use with functions: best done with pointers (later)
    c. Iteration
    d. Pitfalls
        i. Out of bounds index access
        ii. Fixed size
2. Dynamic Arrays
    - Size of an array may not be known at compile time, may vary from run to run
    - Have need to create dynamic arrays:
        - Size is specified at run time
        - Memory is dynamically allocated
        - Memory allocation may fail
        - Memory may need to be managed: once we are done using it, we need to clean up and "free" the memory to be reused
    - Static arrays are automatically allocated and stored on the stack (like regular variables) and automatically destroyed when the stack frame goes out of scope
    - Dynamic arrays are allocated on the heap (preallocated area of memory for the program)

- Dynamic arrays are the norm
- Once created, can be used like static arrays (accessing, assignment, etc.)
- Usage in functions
  - Parameters
  - Return Types
- Pitfall: memory leaks, dangling pointers

3. Multidimensional Arrays
   a. Matrices
   b. Static
   c. Dynamic

Arrays in C

1. Key Points
   a. Arrays as memory addresses
      - Static arrays are *contiguous* memory blocks
      - Array identifiers are references (memory addresses) of the beginning of an array
      - An index is an offset: compiler knows the type and how much memory it takes and is able to compute the memory address of individual elements
      - Alternative syntax is possible, but discouraged
   b. Indexing syntax changes an array into a regular variable: a[0] vs &a[0]
   c. User's responsibility to keep track of array sizes
   d. Syntax for usage in parameters
   e. Dynamic Arrays
      i. Requires a pointer
      ii. Usage of malloc: memory allocation
         1. Input: number of bytes to allocate
         2. Usage of the sizeof() macro
         3. Returns a pointer
         4. Pointer (should) be type casted
      iii. Use the array as normal after creation
      iv. Once done, should use free() to free up the memory
   f. Dynamic Multidimensional Arrays
      i. Requires pointer to pointers
      ii. Memory must be allocated for each "row"
      iii. Memory must be freed in reverse
   g. Pitfalls
      i. ANSI C does not allow a variable length array declaration:
      `int arr[n];`
      To see, compile with: gcc -pedantic -ansi foo.c
      ii. Memory leaks

Arrays in java

1. Syntax
    a. Declaration
    b. Usage (same)
    c. Dynamic allocation: `new` keyword
2. Key differences
    a. No memory management: no need to destroy or free up memory
    b. No pointers: all references use square brackets
    c. Size is maintained through a .length field
    d. `IndexOutOfBoundsException`
    e. Array Utilities: `java.utis.Arrays`:
        i. Searching
        ii. Sorting
        iii. Copying
        iv. Printing
    f. Iterating: enhanced for-loop
3. Alternative: Collections Framework
    a. Dynamic lists: `ArrayList<T>`
    b. Dynamic set (duplicates not allowed, no order): `HashSet<T>`
    c. Instantiation
    d. Usage: add, remove, get(int),

Exercises

1. Create a function to copy an array of integers and return a pointer to the new copy
2. Write a function to reverse the contents of an array
3. Write a function to print an array
4. Write a function to print a matrix
5. Write a function to create a new matrix filled with zeros of a given size
6. Write a function to create the identity matrix of a certain dimension
7. Write a function to compute the addition of two matrices
8. Write a function to compute the average of the elements in an array
9. Write a function to search an array for a particular element and return the index at which it exists
    a. Write a variant function to search an array within a given range
10. Write a function to compute the multiplicity of a given element x within an array (that is, the number of times it appears).
11. Write a function to return a *sorted* copy of an array