CS1010E Lecture 9

Problem Solving with Arrays

Henry Chia (hchia@comp.nus.edu.sg)

Semester 1 2016 / 2017

Lecture Outline

- Random number generation
  - Random number seed
  - Integer and Floating-point sequences
- 1D array problem solving
  - Sorting – selection sort
  - Searching – linear/binary search
- 2D arrays:
  - Declaration and initialization
  - Array element access
  - Array as function argument
  - 2D array problem solving

Random Numbers

- Arrays may be populated with values via multiple user input, or randomly within a specified range, e.g.
  - random integers between 1 and 500
  - random floating-point values between 5.0 and -5.0
- Random numbers generated are equally likely to occur
- `rand` library function (in `stdlib.h`) generates a random integer between 0 and `RAND_MAX`

```
int rand(void);
```

- `RAND_MAX` is a system-dependent integer defined in `stdlib.h`.

Random Numbers

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

#define SIZE 10

void randArray(int x[], int n);
void printArray(int x[], int n);

int main(void) {
   int x[SIZE];

   randArray(x,SIZE);
   printArray(x, SIZE);

   return 0;
}
```

```c
void randArray(int x[], int n) {
   int i;

   for (i = 0; i < n; i++) {
      x[i] = rand();
   }
   return;
}

void printArray(int x[], int n) {
   int i;

   for (i = 0; i < n; i++) {
      printf("%d ", x[i]);
   }
   printf("\n");
   return;
}
```

- What happens each time the program is run?

Random-Number Seed

- To generate different sequences of random values, use the `srand` function (from `stdlib.h`)

```
void srand(unsigned int);
```

- The argument of the `srand` function is an (unsigned) integer, typically known as the seed value
  - For each seed value, `rand` generates a different sequence of random numbers
  - The seed value is typically defined as a constant
- `srand` function is called only once in the program

Random-Number Seed

- Seeding via user-input

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

#define SIZE 10

void randArray(int x[], int n);
void printArray(int x[], int n);

int main(void) {
   int x[SIZE], seed;

   printf("Enter a seed value: ");
   scanf("%d", &seed);
   srand((unsigned int) seed);
   randArray(x,SIZE);
   printArray(x, SIZE);

   return 0;
}
```

- Time-based seeding

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

#define SIZE 10

void randArray(int x[], int n);
void printArray(int x[], int n);

int main(void) {
   int x[SIZE];

   /* time(0) gets current time */
   srand((unsigned int) time(0));
   randArray(x,SIZE);
   printArray(x, SIZE);

   return 0;
}
```

Generating Specific Sequences

- Random integers over a specified range
  - $\{0, 1, 2, 3, 4, 5, 6, 7\}$: `rand()%8`
  - $\{-3, -2, -1, 0, 1, 2, 3\}$: `(rand()%7) - 3`
  - $\{a, a+1, \ldots, b-1, b\}$: `(rand()%(b-a+1)) + a`
- Random floating point numbers over a specified range
  - $[0, 1.0]$: `1.0 * rand()/RAND_MAX`
  - $[0, b]$: `b * rand()/RAND_MAX`
  - $[a, b]$: `((b-a) * rand()/RAND_MAX) + a`

  where a and b are declared `double`
- Generating specific sequences involves *scaling* followed by *shifting*

Sorting Algorithms

- Sorting a group of data values is an operation that is routinely used when analyzing massive amounts of data
- An array $\{x_0, x_1, x_2, \ldots, x_{n-1}\}$ is sorted with respect to a comparator $\preceq$ if and only if $\forall i \in [1, n-1]\ x_{i-1} \preceq x_i$

```
/*
   isSorted returns true if first n elements of array x
   is sorted in ascending order; returns false otherwise.
   Precondition: x[0] to x[n-1] are valid elements of x.
*/
bool isSorted(int x[], int n) {
   int i;
   bool sorted=true;

   for (i = 1; i < n && sorted; i++) {
      if (x[i-1] > x[i]) {
         sorted = false;
      }
   }
   return sorted;
}
```

Selection Sort

- The selection sort routine consists of a number of passes where each pass involves a *selection* and a *swap*
  - Original order:
    5 3 12 8 **1** 9
  - Swap the minimum with the value in the first position:
    1 **3** 12 8 5 9
  - Swap the minimum with the value in the second position:
    1 3 **12** 8 **5** 9
  - Swap the minimum with the value in the third position:
    1 3 5 **8** 12 9
  - Swap the minimum with the value in the fourth position:
    1 3 5 8 **12** **9**
  - Swap the minimum with the value in the fifth position:
    1 3 5 8 9 12
  - Array values are now in ascending order

Selection Sort

```
void selectionSort(int x[], int n) {
   int i, j, minIndex, temp;

   for (i = 0; i < n - 1; i++) { /* n-1 passes */
      minIndex = i;
      for (j = i + 1; j < n; j++) {
         if (x[j] < x[minIndex]) {
            minIndex = j;
         }
      }

      if (minIndex != i) { /* avoid redundant swaps */
         temp = x[minIndex];
         x[minIndex] = x[i];
         x[i] = temp;
      }
   }

   return;
}
```

Search Algorithms

- Another common operation performed with arrays is searching for a specific value
- Searching algorithms fall into two groups:
  - searching an unsorted array
  - searching a sorted array
- By convention, a search function either returns
  - the position of the desired value in the array $\smile$
  - value of -1 if the desired value is not in the array $\frown$

Linear Search

- Consider searching the first n elements of an **unsorted** array x for the value val

```
int linearSearch(int x[], int n, int val) {
   int i = 0, index = -1;

   while (i < n && x[i] != val) {
      i++;
   }

   if (i < n) { /* if (x[i] == val) ??? */
      index = i;
   }

   return index;
}
```

- Perform bounds checking first! The following is risky

```
   while (x[i] != val && i < n)
```

Linear Search

- Consider searching a **sorted** array
- Example: searching for the value 25 in:
  -7 2 14 38 52 77 105
- As soon as the value 38 is reached, conclude that 25 is not in the array

```
int linearSearch(int x[], int n, int val) {
   int i = 0;

   while (i < n && x[i] < val) {
      i++;
   }

   if (i < n && x[i] == val) {
      return i;
   } else {
      return -1;
   }
}
```

Binary Search

- More efficient algorithm for searching a **sorted** array
  - Check the middle of the array and determine if the value is in the first half or second half of the array
  - Within the half array, check the middle and determine if the value is in the first quarter or the second quarter of the array
  - The process of dividing the array into smaller and smaller pieces continues until either
    - the value is found $\ddot\smile$
    - further division is no longer possible $\ddot\frown$

Binary Search

Search for value 14:

-7	2	14	38	52	77	105
l			m			r

$38 > 14$ so choose top half:

-7	2	14	38	52	77	105
l	m	r				

$2 < 14$:

-7	2	14	38	52	77	105
		lmr				

$14 = 14$ so it is found!

Search for value 25:

-7	2	14	38	52	77	105
l			m			r

$38 > 25$ so choose top half:

-7	2	14	38	52	77	105
l	m	r				

$2 < 25$:

-7	2	14	38	52	77	105
		lmr				

$14 < 25$:

-7	2	14	38	52	77	105
		mr	l			

$l > r$ so item is not found!

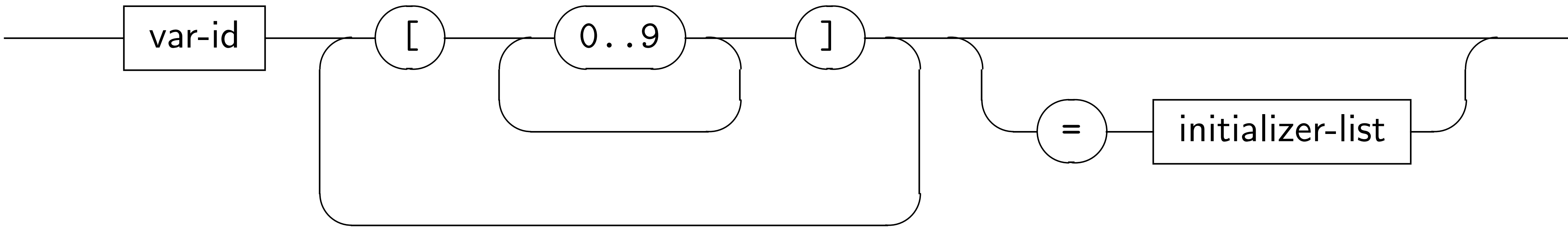
Binary Search

```
int binarySearch(int x[], int n, int val) {
   int left = 0, right = n - 1, mid, index = -1;

   while ((left <= right) && (index == -1)) {
      mid = (left + right) / 2;

      if (x[mid] == val) {
         index = mid;
      } else {
         if (x[mid] > val) {
            right = mid - 1;
         } else {
            left = mid + 1;
         }
      }
   }

   return index;
}
```

2D Array Declaration

- Use 1D array when a set of data values is visualized as a list
- Use 2D array when a set of data values is visualized as a table having both rows and columns

array-decl



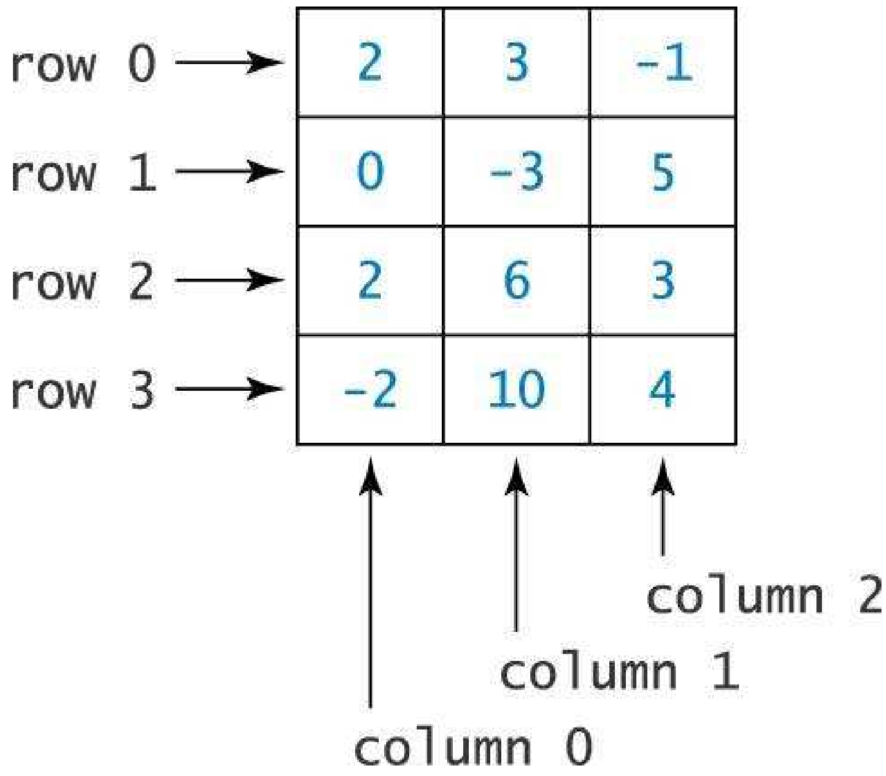
- Example:

```
int t[4][3];
```

- Assuming *row-major* ordering, specify the number of rows followed by the number of columns

2D Array Initialization

```c
int t[4][3]={{ 2, 3,-1},
             { 0,-3, 5},
             { 2, 6, 3},
             {-2,10, 4}};
```



- Initializer used only during declaration
- If the initializing sequence is shorter than the array, the rest of the values are set to zero
- Zero entire array: `int t[4][3]={{0}};`

Accessing 2D Array Elements

- Each element in a 2D array is referenced using an identifier followed by row and column subscripts
- Each subscript value has its own set of brackets
- Each subscript value begins with 0
- In the preceding declaration,
  - value in position `t[1][2]` is 5
  - value in position `t[2][1]` is 6
- Use nested `for` loops to access all elements

```
for (i = 0; i < 4; i++) {
   for (j = 0; j < 3; j++) {
      printf("%4d", t[i][j]);
   }
}
```

2D Array as Function Argument

- When using 2D array as function argument, pass the array together with the number of rows and columns to process
- Note column size in function parameter `int t[][NCOLS]`

```c
#include <stdio.h>

#define NROWS 21
#define NCOLS 11

void print2D(int t[][NCOLS],
             int r, int c);

int main(void) {
   int i, j, t[NROWS][NCOLS];

   for (i = 0; i < NROWS; i++) {
      for (j = 0; j < NCOLS; j++) {
         t[i][j] = i * j;
      }
   }
   print2D(t,NROWS,NCOLS);
   return 0;
}
```

```c
/*
   print2D prints r x c
   table of elements, t
   Precondition: r>=0, 0<= c<NCOLS
*/
void print2D(int t[][NCOLS],
             int r, int c) {
   int i, j;

   for (i = 1; i < r; i++) {
      for (j = 1; j < c; j++) {
         printf("%4d", t[i][j]);
      }
      printf("\n");
   }
   return;
}
```

Example: 2D Cumulative Sum

- The cumulative sum of an element $t_{x,y}$ is the sum of all values above and to the left of it, including itself

$$s_{x,y} = \sum_{i \leq x, j \leq y} t_{x,y} \qquad \mathbf{T} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \Rightarrow \mathbf{S} = \begin{bmatrix} 1 & 3 & 6 \\ 5 & 12 & 21 \\ 12 & 27 & 45 \end{bmatrix}$$

```
#include <stdio.h>
#define SIZE 100

void readData(int t[][SIZE], int r, int c);
void print2D(int t[][SIZE], int r, int c);
void cumulativeSum(int t[][SIZE], int r, int c);

int main(void) {
   int t[SIZE][SIZE]={{0}}, r, c;

   scanf("%d%d", &r, &c);
   readData(t,r,c);
   cumulativeSum(t,r,c);
   print2D(t,r,c);
   return 0;
}
```

Example: 2D Cumulative Sum

```
void readData(int t[][SIZE], int r, int c) {
   int i, j;

   for (i = 1; i <= r; i++) { /* start from row 1 */
      for (j = 1; j <= c; j++) { /* start from col 1 */
         scanf("%d", &(t[i][j]));
      }
   }
   return;
}

void print2D(int t[][SIZE], int r, int c) {
   int i, j;

   for (i = 1; i <= r; i++) {/* start from row 1 */
      for (j = 1; j <= c; j++) {/* start from col 1 */
         printf("%4d", t[i][j]);
      }
      printf("\n");
   }
   return;
}
```

Example: 2D Cumulative Sum

- Use a one-pass strategy:

$$s_{x,y} = t_{x,y} + s_{x-1,y} + s_{x,y-1} - s_{x-1,y-1}$$

```
void cumulativeSum(int t[][SIZE], int r, int c) {
   int i, j;

   for (i = 1; i <= r; i++) { /* start from row 1 */
      for (j = 1; j <= c; j++) { /* start from col 1 */
         t[i][j] = t[  i  ][  j  ] +
                   t[i - 1][  j  ] +
                   t[  i  ][j - 1] -
                   t[i - 1][j - 1];
      }
   }
   return;
}
```

Lecture Summary

- Use of `rand` and `srand` to generate integer or floating-point random sequences
- Understand the technique of a given sorting and searching algorithm; do not memorize code
  - Knowing how the algorithm works will allow you to adapt to different problem scenarios
- Appreciate that 2D arrays are simple extensions of 1D arrays; a 2D array is a 1D array of rows where each row is a 1D array of elements in sequence
  - Each row can thus be passed to a function that accepts a 1D array; however a column cannot be passed this way
- Use of nested loops to access 2D array elements