CS1010E Lecture #9

2D Arrays and Files

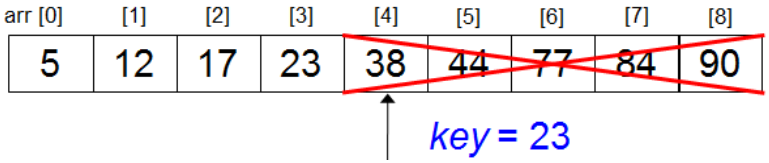
The matrix reloaded

NUS National University of Singapore
Department of Computer Science
School of Computing

Searching

PREVIOUS LECTURE

- Search key
  - The item you are looking for in an array
- Linear search
  - Search the array from one end to the other.
- Binary search
  - Compare arr[mid] with key
  - Each time reduce the scope of search by half.



- Binary search greatly outperforms linear search.

Selection Sort

PREVIOUS LECTURE

- Scan the “alive” array to find the minimum item.
- Swap it with the first item of the “alive” array.

29	**10**	14	37	13
10	**29**	14	37	**13**
10	13	**14**	37	29
10	13	14	**37**	**29**
10	13	14	29	37

10 is the smallest, swap it with the first one, i.e. **29**.

sorted!

Bubble Sort

PREVIOUS LECTURE

- Move the largest item to the end of the array in each iteration by examining neighbours and swap them if they are out of order.

29	**10**	14	37	13
10	**29**	**14**	37	13
10	14	**29**	**37**	13
10	14	29	**37**	**13**
10	14	29	13	37

Pass 1

At the end of the pass 1, the largest item 37 is at the end.

10	**14**	29	13	37
10	**14**	**29**	13	37
10	14	**29**	**13**	37
10	14	13	29	37

Pass 2

At the end of the pass 2, the largest item 29 is at the end of the “alive” array.

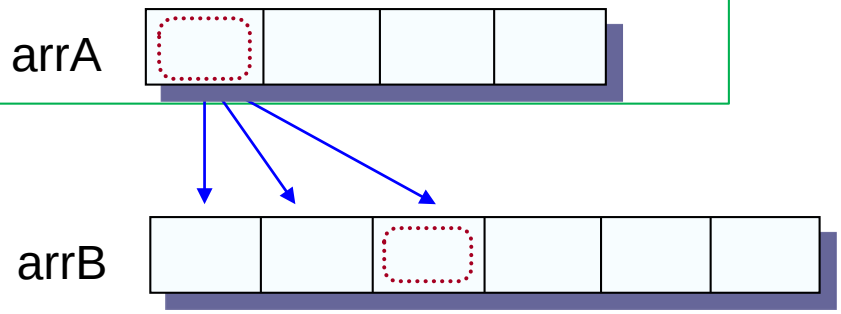
PS 3 Ex #12 Set Containment

```
// Check whether arrA is a subset of arrB.
int is_subset(int arrA[], int sizeA, int arrB[], int sizeB) {

  int i, j, count = 0;
  for (i = 0; i < sizeA; i++) {
    for (j = 0; j < sizeB; j++) {
      if (arrA[i] == arrB[j]) {
        count++;
      }
    } // end inner for loop
  } // end outer for loop

  // if all arrA elements appear in arrB,
  // count should be equal to sizeA
  return sizeA == count;
}
```

Version 1



PS 3 Ex #12 Set Containment

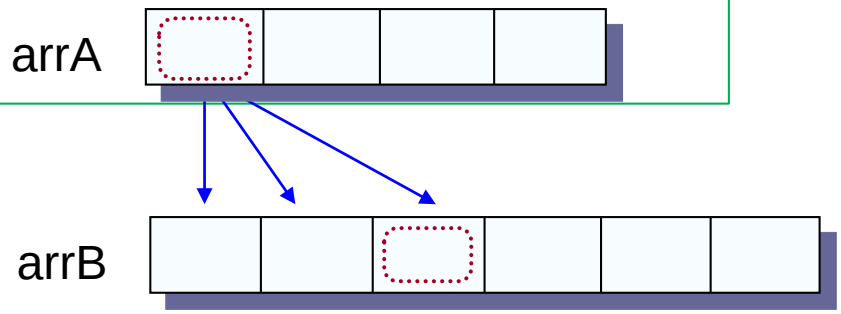
```
// Check whether arrA is a subset of arrB.
int is_subset(int arrA[], int sizeA, int arrB[], int sizeB) {

  int i, j, found = 1;

  for (i = 0; i<sizeA && found; i++) {
    found = 0;
    for (j = 0; j<sizeB && !found; j++) {
      if (arrA[i] == arrB[j]) {
        found = 1;
      }
    } // end inner for loop
  } // end outer for loop

  return found;
}
```

Version 2



Problem Solving Methodology (1/3)

- Start from a hand example (e.g. sample run)
  - Get an idea how to do this example by yourself.
  - Note down all the steps (sub-problems) involved and their sequence.
  - Generalize your idea if necessary.
  - This is just your “algorithm”.

- This step requires good logical thinking ability of you.
  - If you cannot work out the problem by hand, chances are that you won’t be able to solve it on the computer either.
  - Practice and reflection may largely help you.

Problem Solving Methodology (2/3)

- Next, implement your idea (algorithm) into a program.
  - Translate each step into a program fragment (e.g. a function).
  - If a step is quite complex, you may further break it down into several smaller code fragments (e.g. several simpler functions) .
  - Devise the skeleton of the program before typing the code.
    - How many functions to implement?
    - Workflow of the program?

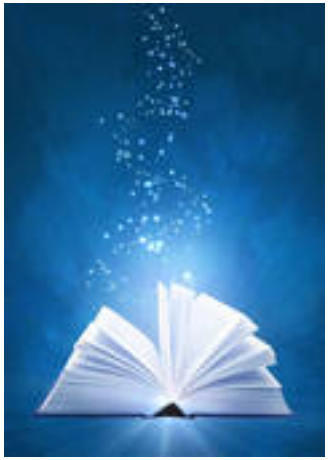
- This step requires your familiarity with syntax of a programming language and knowledge of common programming strategies.
  - A lot of practice is the best way out.
  - Digest the program of others.

Problem Solving Methodology (3/3)

- Finally, verify that your program works correctly.
  - Train yourself of the following good habit:
    - Use vim command "gg=G" to auto-indent your program after coding.
    - Browse through your program one more time.
    - Any messy indentation means your program contains syntax errors (e.g. missing semicolon, extra semicolon, unmatched bracket, etc.)
  - For debugging purpose, strategically insert **`printf()`** statements into your program to print out intermediate values.
    - Check the correctness of intermediate values to narrow down the scope of debugging.
  - Test every branch of the logic in your program.

Learning Objectives

- At the end of this lecture, you should understand:
  - the concept of two-dimensional arrays.
  - how to create and use 2D arrays.
  - the standard procedure to open a file for reading.



Two dimensional Arrays (1/2)

- In C, we may declare a multi-dimensional array by using two or more sets of brackets.
- We only study two-dimensional arrays (2D arrays) in this module, which are useful in representing tabular information.
- Example:

Times Table - 10x10

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	**1**	2	3	4	5	6	7	8	9	10
2	2	**4**	6	8	10	12	14	16	18	20
3	3	6	**9**	12	15	18	21	24	27	30
4	4	8	12	**16**	20	24	28	32	36	40
5	5	10	15	20	**25**	30	35	40	45	50
6	6	12	18	24	30	**36**	42	48	54	60
7	7	14	21	28	35	42	**49**	56	63	70
8	8	16	24	32	40	48	56	**64**	72	80
9	9	18	27	36	45	54	63	72	**81**	90
10	10	20	30	40	50	60	70	80	90	**100**

Two dimensional Arrays (2/2)

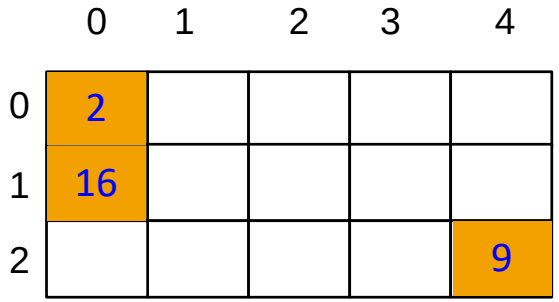
SYNTAX

<data type> <array variable>[<rows>] [<cols>];

- Example:

```
// array with 3 rows, 5 columns
int a[3][5];
a[0][0] = 2;
a[2][4] = 9;
a[1][0] = a[2][4] + 7;
```

Row number *Column number*



2D Array Initializers

```
// nesting one-dimensional initializers
int a[3][5] = { {4, 2, 1, 0, 0},
                {8, 3, 3, 1, 6},
                {0, 0 ,0, 0, 0} };


// partial initialization, implicit zeros
int b[3][5] = { {4, 2, 1},
                {8, 3, 3, 1, 6} };
```

Array b:

	0	1	2	3	4
0	4	2	1	0	0
1	8	3	3	1	6
2	0	0	0	0	0

Q: What about those unmentioned elements?

Passing 2D Arrays to Function Calls

```c
#include <stdio.h>

int sum_array(int arr[3][5], int rows, int cols);

int main(void) {
  int foo[3][5] = { {3,7,1}, {2,1}, {4,6,2} };
  printf("sum is %d\n", sum_array(foo, 2, 5));
  return 0;
}

// sum some elements in arr
int sum_array(int arr[3][5], int rows, int cols) {
  int i, j, total = 0;
  for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
      total += arr[i][j];
    }
  }
  return total;
}
```

Caller specifies array to pass

*Give **the same array** a new name*

Q: What is the output?

sum = 14

Passing 2D Arrays to Function Calls

- Column number must be specified in function header/ prototype.

```
int sum_array(int arr[3][5], int rows, int cols) {
  int i, j, total = 0;
  for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
      total += arr[i][j];
    }
  }
  return total;
}
```

Row number may be omitted

```
int sum_array(int arr[ ][5], int rows, int cols) {
  int i, j, total = 0;
  for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
      total += arr[i][j];
    }
  }
  return total;
}
```

However, column number must be present

Passing a Row of 2D Array to Function

- In C, a 2D array is actually an array of arrays where each row is an array.
- Example:

```
// ...
int main(void) {
  int arr[][] = { { 1, 2, 3, 4 }, { 5, 6, 7, 8 },
                  { 0, 0, 0, 0 } };
  print_array(arr[0], 4);
  return 0;
}

void print_array(int arr[], int size) {
  int i;
  for (i = 0; i < size; i++) {
    printf("%d\n", arr[i]);
  }
}
```

Passing one row of 2D array

Q: What is the output?

```
1
2
3
4
```

Demo #1 : Matrix Addition (1/2)

- A two-dimensional array where all the rows have the same number of elements is also known as a matrix because it resembles that mathematical concept.
- To add two matrices, both must have the same size (i.e. same number of rows and columns).
- To compute C = A + B, where A, B, C are matrices:

$$c_{m,n} = a_{m,n} + b_{m,n}$$

- Example on 2 × 4 matrices:

$$\begin{pmatrix} 10 & 21 & 7 & 9 \\ 4 & 6 & 14 & 5 \end{pmatrix} + \begin{pmatrix} 3 & 7 & 18 & 20 \\ 6 & 5 & 8 & 15 \end{pmatrix} = \begin{pmatrix} 13 & 28 & 25 & 29 \\ 10 & 11 & 22 & 20 \end{pmatrix}$$

A B C

Demo #1 : Matrix Addition (2/2)

```
// Add mtxA and mtxB to obtain mtxC and return it
void add_mtx(int mtxA[MAX_ROW][MAX_COL], int mtxB[MAX_ROW][MAX_COL],
             int mtxC[MAX_ROW][MAX_COL], int row_size, int col_size) {

  int row, col;

  for (row = 0; row < row_size; row++) {
    for (col = 0; col < col_size; col++) {
      mtxC[row][col] = mtxA[row][col] + mtxB[row][col];
    }
  }
}
```

physical size

actual size to process

$$\underbrace{\begin{pmatrix} 10 & 21 & 7 & 9 \\ 4 & 6 & 14 & 5 \end{pmatrix}}_{A} + \underbrace{\begin{pmatrix} 3 & 7 & 18 & 20 \\ 6 & 5 & 8 & 15 \end{pmatrix}}_{B} = \underbrace{\begin{pmatrix} 13 & 28 & 25 & 29 \\ 10 & 11 & 22 & 20 \end{pmatrix}}_{C}$$

Demo #2 : Min and Max

This is Problem Set 3 Ex #21 on CodeCrunch

- Write a program to print out the minimum and maximum elements of an integer 2D array.
- Your program should contain a function

```
void get_min_max(int mtx[MAX_ROWS][MAX_COLS],
    int num_rows,  int num_cols, int *min_p, int *max_p)
```

  that returns the minimum and maximum elements through two pointers.

- Sample run:

```
Enter the size of the matrix: 4 5
Enter elements row by row:
1 0 1 0 1
0 1 0 1 0
1 0 1 0 1
0 1 0 1 0
Min = 0
Max = 1
```

Input Re-direction (1/2)

- Test case #7 of Exercise #21:

```
Enter the size of the matrix: 9 10
Enter elements row by row:
1 1 1 3 1 0 0 7 6 3
0 9 2 5 3 6 8 3 1 6
9 5 8 3 1 3 2 0 4 9
4 9 5 2 9 6 3 3 9 6
8 2 1 0 0 1 2 3 4 6
1 2 4 6 7 8 4 2 1 7
1 3 5 7 9 8 2 4 9 1
1 9 2 5 3 6 8 4 2 6
8 2 0 1 8 2 9 3 8 1
Min = 0
Max = 9
```

It's tedious and error-prone to manually key in all data from keyboard.

Input Re-direction (2/2)

- It's inconvenient for user to key in large amount of data for array.
- We may store input data to a program in a text file (e.g. min_max7.in) and let the program read data from the file instead.
  - A simple way is to use input redirection, a feature provided by (UNIX and Windows) operating system.
    - On sunfire, give a try of the following command:

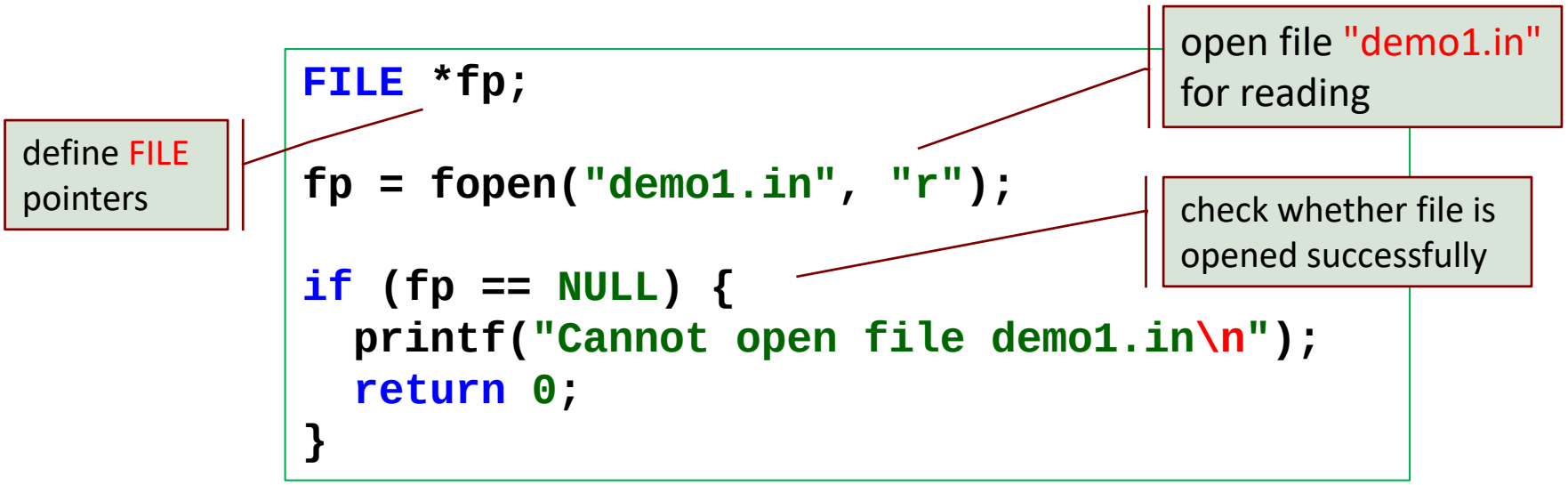
```
a.out < min_max7.in
```

  - Another way is file processing provided by C language.

File Processing : Overview

- C provides functions to handle file processing.
- Typically, file processing requires:
  1) Opening a file (fopen)
  2) Testing if the file is opened successfully
  3) Reading data from or writing data to the file (fscanf, fprintf)
  4) Closing the file (fclose)
- Here we only study how to open files for reading. You may explore writing data to files by yourself, if interested.
- All I/O functions are defined in the header file <stdio.h>

File Processing : Opening a File



- fopen() function opens a file.
  - It returns **NULL** if an error is encountered.
  - Otherwise, it returns a pointer pointing to the first data in the file.
- Error can happen, e.g. when you try to open a file for reading, but the file doesn't exist in the specified directory, or you don't have permission to open this file.

File Processing : Reading from a File

```
int i, var;
FILE  *fp;

fp = fopen("demo1.in", "r");
if (fp == NULL) {
  return 0;
}
for (i = 0; i < 4; i++) {
  fscanf(fp, "%d", &var);
  printf("%d\n", var);
}
```

read from file

Input file: demo1.in

```
10 20 30 40
```

```
10
20
30
40
```

- fscanf() function reads data from a file.
- Its usage is similar to scanf()
  - except for the additional FILE * pointer argument

File Processing : Closing a File

```c
#include <stdio.h>
int main(void) {

  int  i, var;
  FILE *fp;

  fp = fopen("demo1.in", "r");
  if (fp == NULL) {
    printf("Cannot open file demo1.in\n");
    return 0;
  }

  for (i = 0; i < 4; i++) {
    fscanf(fp, "%d", &var);
    printf("%d\n", var);
  }

  fclose(fp);
  return 0;
}
```

close the file pointed to by this pointer

Today's Summary

Arrays II

Two dimensional arrays

- Syntax and usage

Unix input (and output) direction

Simple file processing

- open -> read -> close