EEE101: C Programming & Software Engineering I

Lecture 7: Arrays and Pointers 2

Dr. Rui Lin/Dr. Mark Leach
Office: EE512/EE510
Email: rui.lin/mark.leach@xjtlu.edu.cn
Dept. of EEE XJTLU

Outline of Today's Lecture (7)

- Passing arrays to functions
- Pointer operations/Pointer compatibility
- Writing array-processing functions
- Pointer arrays vs. Multidimensional arrays
- Command line arguments
- Pointer to functions

Passing Arrays to Functions (1/2)

- C automatically passes arrays to functions using call by reference (each element value could be modified in the calling function)
- Unlike simple variables that are call by value

Arrays – An Introduction (2/2)

```
#include <stdio.h>
void modifyArray(int [], int); /*pointer to array*/
int main() {
int array[3]={10,20,30}, i;
printf("Original values in array\n");
for(i=0; i<3; i++)
        printf("%3d", array[i]);
modifyArray(array, 3);      /*Call by reference*/
printf("New values in array\n");
for(i=0; i<3; i++)
        printf("%3d", array[i]);}
```

Variable Interchange (swapping)

- How could you swap the values stored in two variables? **With a temporary variable**

```
#include <stdio.h>
int main(){
int x=5,y=10;
swap(&x,&y);                    /*send addresses*/
}
void swap(int *u, int *v){          /*u and v are pointers*/
        int temp;
        temp = *u;
        *u = *v;          /*modifying *u,*v modifies x,y*/
        *v = temp;}
```

Protecting Array Contents (1/3)

- Remember function arguments can be passed:
  - **By value** - **only the value is sent to the function and the argument cannot be changed in the function**
  - **By reference** - **pointer or address is sent to the function and the argument can be changed in the function**
- This means the original value is protected when only the value is sent to the function.
- Arrays can **only** be passed by **reference** so…

How do you prevent the original array content from being modified by a function?

Protecting Array Contents (2/3)

- To prevent modification of call by reference arguments the qualifier **const** can be used for the receiving variable declarations

```
void modifyArray(const int a[], int size){
        int i;
        for(i=0; i<size;i++)
                a[i]+=2;        /*error*/
        }
```

Protecting Array Contents (3/3)

- Another example:

```
int sum(const int ar[], int n){
        int total=0;
        for(int i=0; i<n ;i++)
                total+=ar[i];                    /*valid*/
        return total;
        }
```

Constant Pointers

- How about a constant pointer?

```
int x, y;
int * const ptr = &x;
*ptr = 7;                /*Assign a new value to x*/
ptr = &y;                /*Error – can't change address*/
```

If a pointer is declared and initialised as constant the address contained in the pointer cannot be changed

So…what does **int const** *p = # mean?

Pointer Compatibility (1/2)

- When assigning pointer to other pointers, they must be the same type including **const**
- You cannot assign a non-**const** pointer to a **const** or to a pointer to a **const** because a non-**const** pointer would allow you to change the value of the **const**

```
int x, y;
int const * ptr1 = &x;
int * ptr2 = &y;
ptr2 = ptr1;            /*Error ptr1 is constant*/
```

Pointer Compatibility (2/2)

- You cannot use a pointer declared to point to a **const** variable to change the variable value (even if the variable is not a **const**)

```
int x;
int const y;
int const *ptr1 = &x;
int  *ptr2 = &y;        /*Error ptr2 could change const y*/
x=5;                    /*This is ok x is not const*/
*ptr1=6;                /*Error ptr1 points to a const*/
```

Arrays as Function Arguments (1/6)

- When writing an array processing function

What should be sent to the function?

The address of the first element of the array

What is the address of first element of an array ar?

You can use ar or &ar[0]

Anything else?

How about the size of the array so you know how many elements to process

Arrays as Function Arguments (2/6)

Example of a function able to sum all array elements

```
int sum(int*ar, int n){
int i, total=0;
for(i=0; i<n, i++)
        total+=ar[i];
return total;
}
```

Arrays as Function Arguments (3/6)

- The declaration **int** ar[] as a formal function parameter e.g.

```
int sum(int ar[], int n);
                    /*function prototype example*/
```

Use of this style makes it very clear that the function is processing a one-dimensional array.

Arrays as Function Arguments (4/6)

- The following function prototypes are equivalent

```
int sum(int *ar, int n);
int sum(int *, int);
int sum(int ar[], int n);
int sum(int [], int n);
```

- The function prototype doesn't require a variable name, but the function definition does

Arrays as Function Arguments (5/6)

- Two pointer variables could be used to describe a one-dimensional array
  - **One pointer is the array name (address of first element)**
  - **Second pointer is the first memory location after the last array element. C guarantees this is a valid address.**
- How could these be applied?
  - **Move the first pointer through the array using pointer operations**
  - **Compare the pointer values to decide when to stop**

Example…

Arrays as Function Arguments (6/6)

```
#include <stdio.h>
#define SIZE 5
int sum(int *, int *);                  /*function prototype*/
main() {
int hours[SIZE] = {3,20,13,21,18}, total;
total = sum(hours, hours + SIZE);                  /*function call*/
}
int sum(int *start, int *stop) {
        int total = 0;
        while ( start != stop ) {
                total += *start;                   /*add value to sum*/
                start++;          /*advance pointer to next element*/
                }
        return total;
}
```

Pointer to an array….

- So we can use pointers to deal with 1D arrays…
- What happens when we have a 2D array?

```
int table[4][3];    /*declares a 2D array*/
```

We can think of this as a 4 element array, where each element is another 3 element array

How would we declare a pointer to this table?

```
int (*ptable)[3]; /*declares a pointer called ptable*/
```

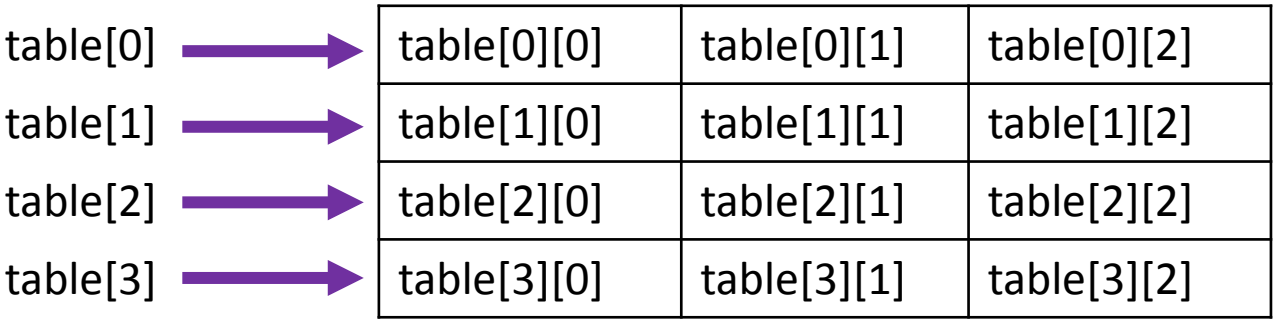
The pointer points to 3 int's (i.e. to an array)

Note the ( ) are used as [] has higher precedence than *

2D Arrays and Pointers (1/5)

int table[4][3] **/*has 4 elements*/**

/*one element is a 3 element array*/



table[0] is the address &table[0][0]

Then (table[0]+2) is the address &table[0][2]

And (table[2]+1) is the address &table[2][1]

Using the * these elements can be accessed

2D Arrays and Pointers (2/5)

int table[4][3] **/*has 4 elements*/**

/*one element is a 3 element array*/

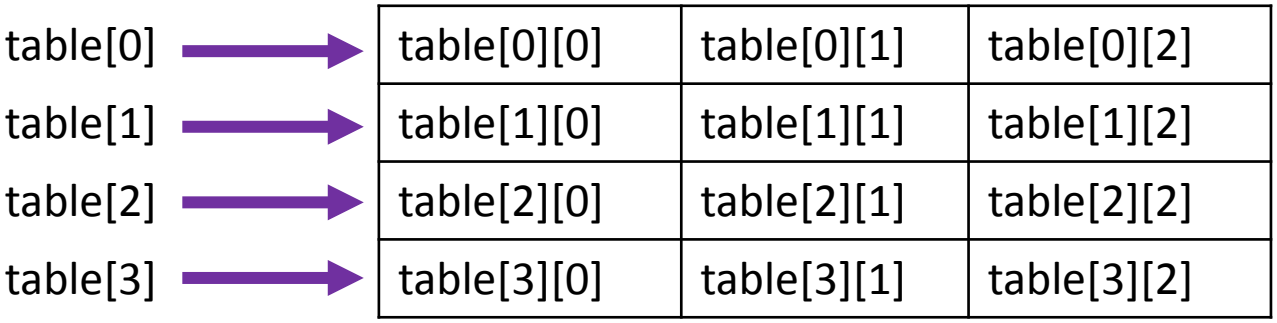


table is also the address &table[0][0] but points to the whole first row

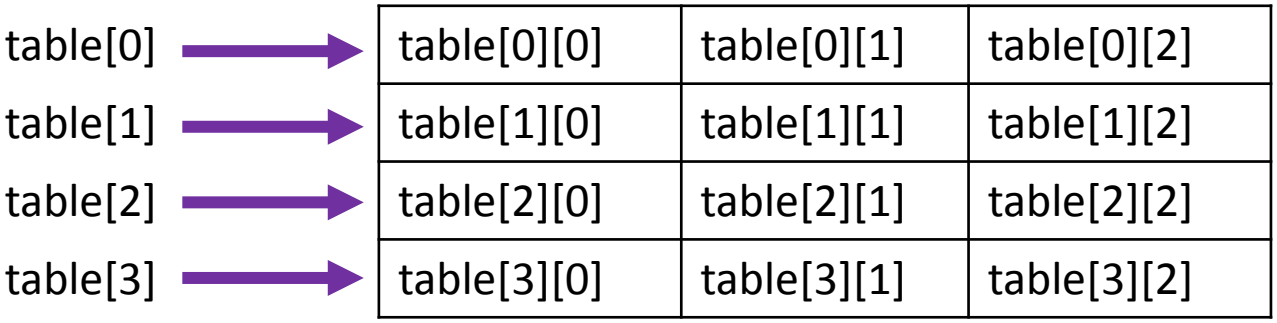
table+2 is now the address &table[2][0]

How would you address &table[2][1]?

2D Arrays and Pointers (3/5)

int table[4][3] **/*has 4 elements*/**

/*one element is a 3 element array*/



How would you address &table[2][1]?

***(table+2)+1**

How would the element table[2][1] be accessed?

***(*(table+2)+1)**

2D Arrays and Pointers (4/5)

Addressing Summary

Location(s) Addressed	Addressing
x[0]([0],[1],…,[N])	x variable name points to address of first element (where the first element is an array)
x[0]([0],[1],…,[N])(	(*p)[N] a pointer to an array
x[0][1]	(*x)+1
x[1][0]	x[1] address of first element of second array x[0]+1

2D Arrays and Pointers (5/5)

Accessing Summary

Location(s) Addressed	Accessing
x[0][0]	*((*x)+0)(or just **x)
x[0][1]	*((*x)+1) (or just *(*x)+1)
x[1][0]	*(*(x+1)+0) (or just **(x+1))

Quick Quiz 1

Consider the following:

```
int table[4][3] ;
int *ptr;
```

Which of the following commands addresses the second row of the array table?

a) ptr = table[1];
b) ptr = table[1][2];
c) ptr = table[2];
d) ptr = table+1;
e) None of the above

Quick Quiz 1

Consider the following:

```
int table[4][3] ;
int *ptr;              /*(*ptr)[3]*/
```

Which of the following commands addresses the second row of the array table?

a) **ptr = table[1]; /*points to the first element of*/**

b) ptr = table[1][2]; **/*second row only*/**

c) ptr = table[2];

d) **ptr = table+1; /*points to row, but ptr should be*/**

e) None of the above **/*changed as above*/**

Arrays of pointers (1/2)

So we have looked at arrays and how they relate to pointers. But now…how about an array of pointers?

```
int *table[3];
```

This is an array with 3 elements and each element is an **int** pointer

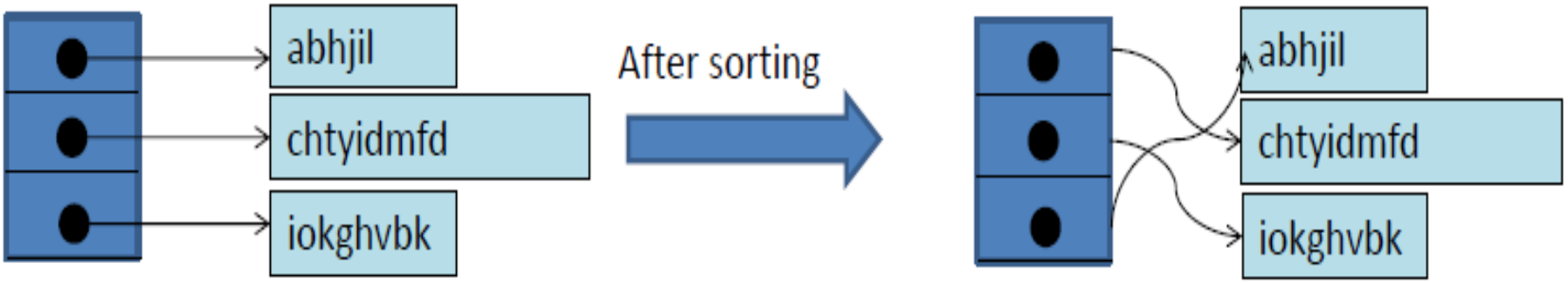
```
int (*table)[3];
```

This is **single pointer** that points to an **int** array with 3 elements

Arrays of pointers (1/2)

Why would we want an array of pointers?

- **Better use of memory space**
- **Processing efficiency**



Quick Quiz 2

Which of the following commands declare and initialise an array containing the days Saturday and Sunday?

a) **char** *days = {“Saturday”, “Sunday”};

b) **char** *days[2] = {“Saturday”, “Sunday”};

c) **char** days[2] = {“Saturday”, “Sunday”};

d) None of the above

Quick Quiz 2

Which of the following commands declare and initialise an array containing the days Saturday and Sunday?

a) **char** *days = {“Saturday”, “Sunday”};

b) **char *days[2] = {“Saturday”, “Sunday”};**

c) **char** days[2] = {“Saturday”, “Sunday”};

d) None of the above

There's more…Pointer to Pointer

How do we declare a pointer to another pointer?

```
int *p1;     /*a pointer to an int named p1*/
int **p2;    /*a pointer to an int pointer named p2*/
p2=&p1;      /*so p2 could hold the address of p1*/
```

For sending and returning pointers to functions

Remember-The pointer and array should be matched at the same level:

2D Array ⟷ pointer to pointer

1D Array ⟷ pointer

Examples

```
int table[4][3], myarray[4][3];
int *table1[3];
int **pTable;
int *Ptr1;
int (*pPtr)[3];
```

```
*pTable=table[1]; /*Valid, table[1] indicates the array of 2nd row*/
Ptr1=table+1; /*Invalid, the level of pointer and array should match*/
pTable=table1;          /*Valid*/
Ptr1=table[1];          /*Valid*/
Ptr1 =*(table+1);       /*Valid*/
pTable=table[1];        /*Invalid*/
table= myarray;         /*Invalid*/
pPtr =table;            /*Valid*/
```

Some other observations

Consider the following:

```
float table[10][20];
```

In general (for **n**th element of **m**th array)

```
*(*(table+m)+n) == table[m][n]
```

```
(*(table+m))[n] == table[m][n]
```

Functions & multi-dimensional Arrays

We have seen how to deal with 1D arrays and pointers so that they can be passed to a function

For a 1D array for example, the first element address is sent to a pointer and the length of the array

What is needed for a multidimensional array?

and

How would the prototype be declared?

2D Array Function Prototypes

```
void sum(int ar[][3], int rows);
```

```
void sum(int (*ar)[3], int rows);
```

```
void sum(int **ar, int rows int cols);
```

Example 1-1 (Function main)

```
#include <stdio.h>
#define COLS 2
#define ROWS 3
double sum2d(double (*ar)[COLS], int ); /*prototype*/
main () {
double ar[ROWS][COLS] = { 1.2, 3.2, 4.9, 3.0, 23.9, 18.7 };
double total;
total = sum2d(ar,ROWS);          /*call function*/
printf("Sum of all elements is %lf",total);
}
```

Example 1-2 (Function sum2d)

```
double sum2d(double (*ar)[COLS], int rows) {
int i,  j;
double tot = 0;
for(i=0; i<rows; i++)          /*nested loops for 2d array*/
        for(j=0; j<COLS; j++)
                tot += ar[i][j];  /*sum elements one by one*/
return tot;
}
```

Command-line Arguments (1/3)

We sometimes want to pass arguments into a program (i.e. into function main) when it begins executing i.e. from the command-line.

For example, we want run a program (read_file) to read a certain number of lines (1000) from a file (student_record) into the memory space and we want to run this program by typing on the command-line

```
read_file student_file 1000
```

How does this work?

Command-line Arguments (2/3)

In C, when main is called it has 2 arguments. (These can be ignored if you are not using them.)

```
main(argc, argv)
```

argc is the **int** number of command line arguments.

argv is a pointer to an array of strings where the arguments will be stored

Notes:

argv[0] is the name of the program and so argc is always at least 1.

Also argv[argc] = NULL

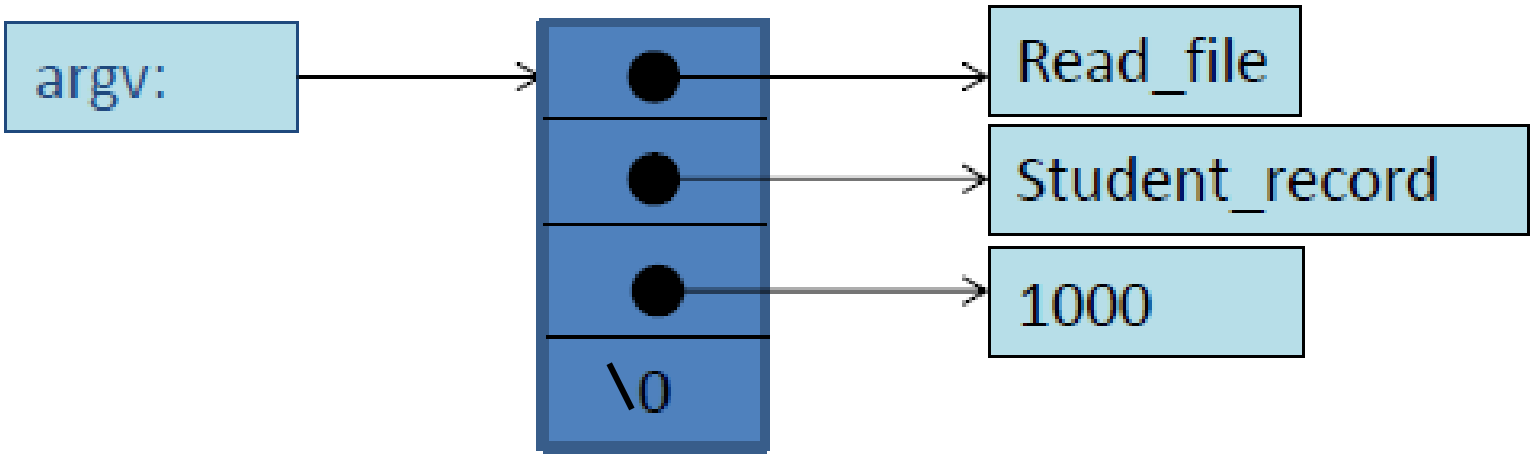
Command-line Arguments (3/3)

In the example:

```
read_file student_file 1000
```

What does argc equal?

argc=3



Pointers to Functions (1/3)

Just when you thought it was safe…

A function is not a variable, however, because functions are stored at memory locations they have addresses.

Why would we point to a function?

Sometimes we want to use different functions that take the same arguments depending on a condition

For example, choosing to add two numbers or subtract them. The add and subtract functions would take and return the same arguments. A single function call could be written to point to the desired function.

Example…

Pointers to Functions (2/3)

Simple example syntax:

Firstly we have a function such as:

```
int addint(int n, int m) {return m+n;}
```

Now we define a function pointer which receives 2 **int**'s and returns **int**:

```
int (*Paddint)(int, int);
```

Now the function can be pointed to:

```
Paddint = &addint; /*& is optional here*/
```

Now we can use the pointer to call the function:

```
int sum = (*Paddint)(2,3); /*same as addint(2,3)*/
```

Pointers to Functions (3/3)

```
#include <stdio.h>
int add(int x, int y);                                  /*function prototypes*/
int subtract(int x, int y);
int domath(int (*mathop)(int, int), int x, int y);

int main() {
int a = domath(add, 10, 2);
int b = domath(subtract, 10, 2);
printf("Subtract gives: %d\n", b);
printf("Add gives: %d\n", a);}

int add(int x, int y) { return x + y; }                 /*function definitions*/
int subtract(int x, int y) { return x - y; }
int domath(int (*mathop)(int, int), int x, int y) { return (*mathop)(x, y);}
```

Quick Question?

If

```
int (*Paddint)(int, int);
```

is a pointer to a function…What is this?

```
int *Paddint(int, int);
```

Questions?

Remember the labs really are important ☺