CPT109: C Programming & Software Engineering I

Lecture 6: Arrays and Pointers 2

Dr. Xiaohui Zhu
Office: SD535
Email: Xiaohui.zhu@xjtlu.edu.cn

Outline of Today's Lecture (6)

- Pointers – Quick Review
- Call by Reference
- Passing arrays to functions
- Protecting arrays/Pointer compatibility
- Multidimensional arrays
- Arrays of Pointers
- Pointers to Pointers

Pointers – Quick Review

- Pointers are variables that are used to store memory addresses.
- A pointer is used to store the address of another variable – of the same type.
- A pointer can be used to get values from or send values to the variable they point at.
- A pointer to an array can be incremented or decremented to move through the elements of an array.

Functions - Call by Reference

- Previously values stored in variables have been sent to functions call by value
- This means that the variable value itself cannot be changed
- Pointers allow access to a variable declared in one function to be changed within another function call by reference
- Let’s see how we could use a function to swap values in variables local to the calling function…

Variable Interchange (swapping)

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

```
#include <stdio.h>
void swap(int *u, int *v);
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;}
```

Passing Arrays to Functions (1/9)

- 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

Passing Arrays to Functions (2/9)

- 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

Passing Arrays to Functions (3/9)

```
#include <stdio.h>
void modifyArray(int *a, int); /*pointer to array*/
int main() {
int array[3]={10,20,30}, i;
printf(“\nOriginal values in array\n”);

for(i=0; i<3; i++)
        printf(“%d ”, array[i]);
modifyArray(array, 3);      /*Call by reference*/
printf(“New values in array\n”);

for(i=0; i<3; i++)
        printf(“%d ”, array[i]);}
```

Passing Arrays to Functions (4/9)

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

Arrays as Function Arguments (5/9)

Another example of a function processing an array - sum all array elements

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

```
int sum(int *ar, int size);
```

Requires prototype

Arrays as Function Arguments (6/9)

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

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

Equivalent to **int** *ar

Use of the style **int** ar[] makes it very clear that the function is processing a one-dimensional array.

Arrays as Function Arguments (7/9)

- 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 (8/9)

- 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 (9/9)

```
#include <stdio.h>
#define SIZE 5
int sum(int *start, int *stop);            /*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 total*/
                start++;         /*advance pointer to next element*/
                }
        return total;
}
```

Protecting Array Contents (1/6)

- Remember function arguments can be passed either:
  - **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**
- 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 (2/6)

- 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 (3/6)

- Another example:

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

Constant Pointers (4/6)

- 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

Pointer Compatibility (5/6)

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

Pointer Compatibility (6/6)

- When assigning a 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**
- a non-**const** pointer would allow you to change the value of the **const**

```
int x;
const int * ptr1 = &x;
int * ptr2;
ptr2 = ptr1;          /*Error ptr1 is a pointer to a const*/
```

Pointer to an array…. (1/7)

- 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

1D Arrays and Pointers (2/7)

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

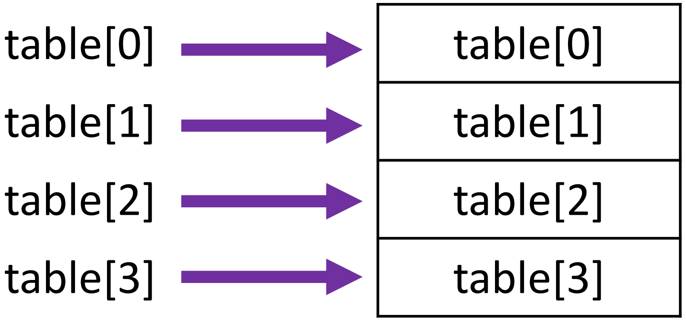


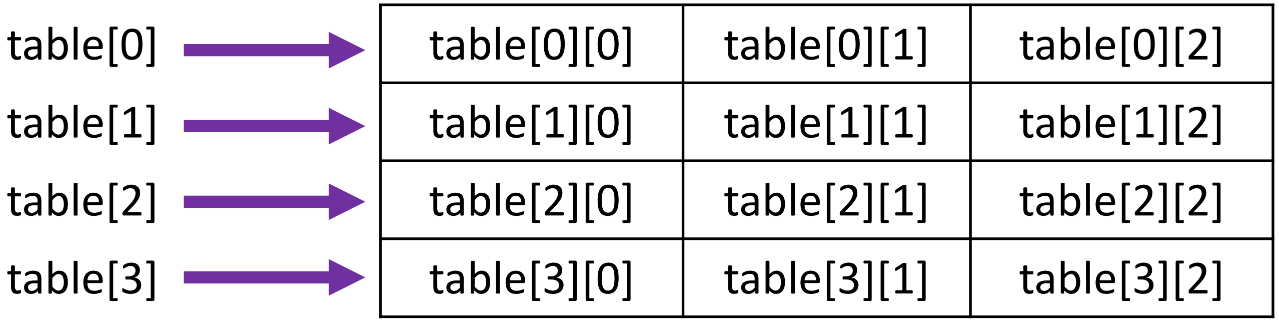
table is the address &table[0]

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

Using the '*' the element can be accessed: *(table+2)

2D Arrays and Pointers (3/7)

```
int table[4][3]    /*has 4 elements*/
                   /*one element is a 3 element array*/
```



table[0]	table[0][0]	table[0][1]	table[0][2]
table[1]	table[1][0]	table[1][1]	table[1][2]
table[2]	table[2][0]	table[2][1]	table[2][2]
table[3]	table[3][0]	table[3][1]	table[3][2]

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

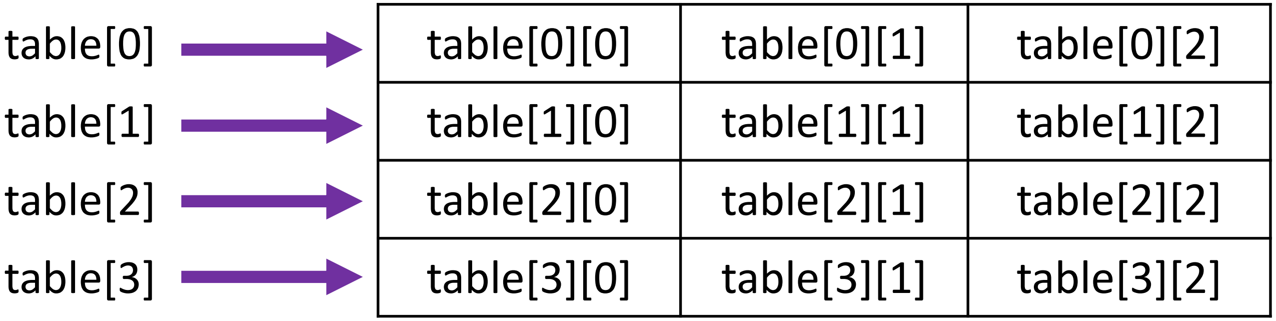
Then (table[0]+2) is the address &table[0][2]

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

Using the * these elements can be accessed

2D Arrays and Pointers (4/7)

```
int table[4][3]     /*has 4 elements*/
                    /*one element is a 3 element array*/
```



table[0] →	table[0][0]	table[0][1]	table[0][2]
table[1] →	table[1][0]	table[1][1]	table[1][2]
table[2] →	table[2][0]	table[2][1]	table[2][2]
table[3] →	table[3][0]	table[3][1]	table[3][2]

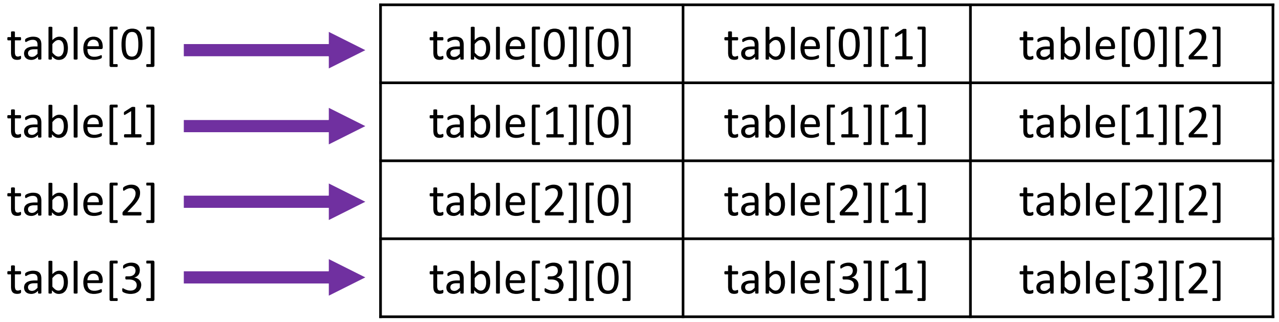
How would we declare a pointer to table[0]?

```
int *ptable;        /*declares a pointer called ptable*/
```

The pointer points to 1 int (since table[0] is 1 int)

2D Arrays and Pointers (5/7)

```
int table[4][3]    /*has 4 elements*/
                   /*one element is a 3 element array*/
```



table[0]	table[0][0]	table[0][1]	table[0][2]
table[1]	table[1][0]	table[1][1]	table[1][2]
table[2]	table[2][0]	table[2][1]	table[2][2]
table[3]	table[3][0]	table[3][1]	table[3][2]

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

but points to the whole first row

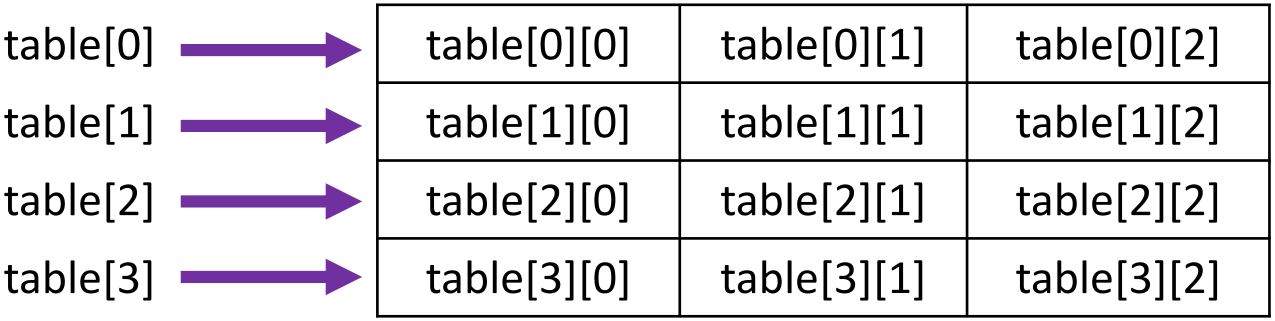
table+2 is now the address &table[2][0]

but again points to the whole third row

How would you address &table[2][1] in this form ?

2D Arrays and Pointers (6/7)

```
int table[4][3]     /*has 4 elements*/
                    /*one element is a 3 element array*/
```



table[0] →	table[0][0]	table[0][1]	table[0][2]
table[1] →	table[1][0]	table[1][1]	table[1][2]
table[2] →	table[2][0]	table[2][1]	table[2][2]
table[3] →	table[3][0]	table[3][1]	table[3][2]

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

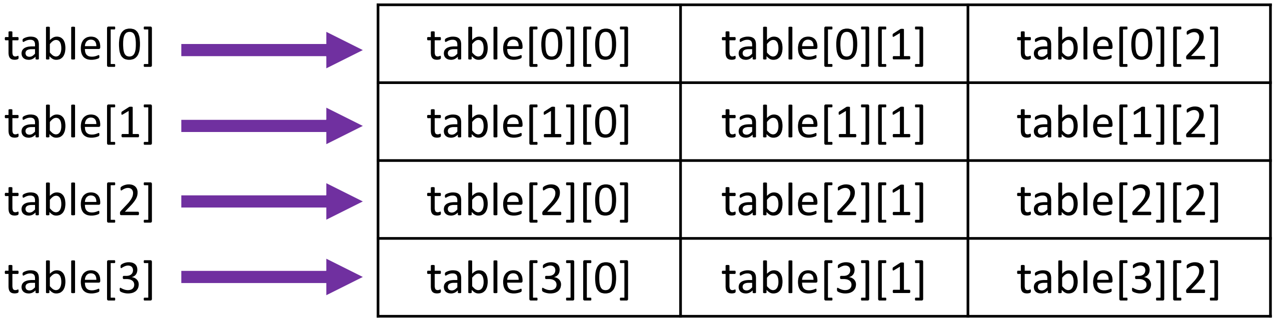
```
*(table+2)+1     /* *(table+2) same as &table[2][0]*/
```

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

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

2D Arrays and Pointers (7/7)

```
int table[4][3]     /*has 4 elements*/
                    /*one element is a 3 element array*/
```



table[0] →	table[0][0]	table[0][1]	table[0][2]
table[1] →	table[1][0]	table[1][1]	table[1][2]
table[2] →	table[2][0]	table[2][1]	table[2][2]
table[3] →	table[3][0]	table[3][1]	table[3][2]

How would we declare a pointer to 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 *

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]
```

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);
```

Example 1-1 (Function main)

```
#include <stdio.h>
#define COLS 3
#define ROWS 2
double sum2d(double (*ar)[COLS], int rows); /*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)[3], 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;
}
```

Equivalent expressions:

`tot += *(*(ar+i)+j);`   or   `tot += *(ar+i)[j];`

Quick Quiz 1

Consider the following:

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

Which of the following instructions 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];** /* row 2 */

b) ptr = table[1][2]; /* row 2 column 3 */

c) ptr = table[2]; /*row 3*/

d) **ptr = table+1;** /*row 2*/

e) None of the 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

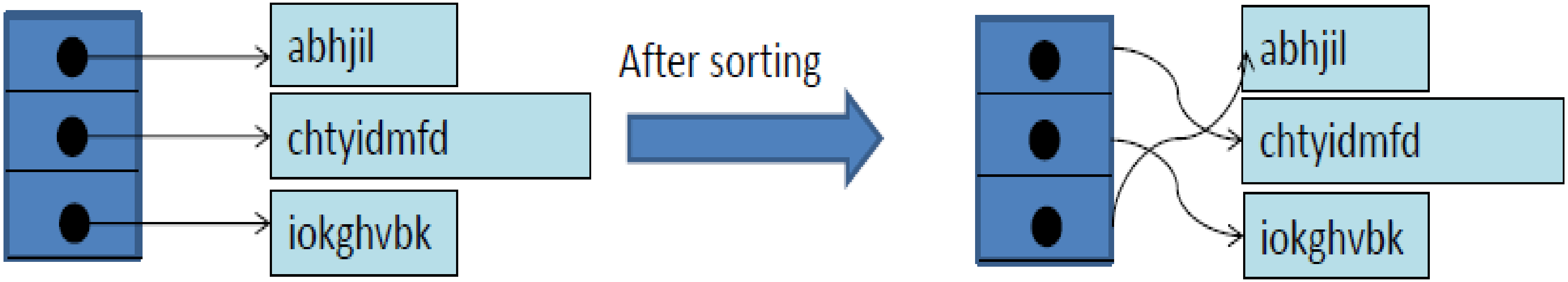
```
int (*table)[3];
```

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

Arrays of pointers (2/2)

Why would we want an array of pointers?

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



Quick Quiz 2

Which of the following commands declares and initialises 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

Notes for lab test (Week 7)

The lab test will be hold in labs **between 15:30 and 17:30 on 1 November 2023 (Wednesday).**

All students should **attend this onsite test**. Attendance will be checked during the lab time.

Students will be arranged into different labs according to their programmes. **Please check the "online continuous assessment" Section in LMO to find your lab position**.

Similar to continuous assessment , the test is also based on LMO and will be graded by CodeRunner.

Questions?

Remember the labs really are important ☺