

# Arrays and Pointers

CS2263 – Systems Software Development

1

## Learning Outcomes


At the conclusion of this lecture students should be able to:

- Explain how pointers can interact with the call stack (pointers as function arguments).
- List two pointer-use “fails” and for each case, explain why
- Explain the relationship between pointers and arrays
- Program using arrays and pointers

2

Opening Thoughts

Thanks to Brandon Ogon



\*Local variable\*

What about the place outside the curly braces.

\*Compiler\*

That's beyond your borders.

3

Your boss  
will give you  
arrays

4

Me and my friends  
thinking about  
pointers



5

## References

- Lu, Yung-Hsiang. 2015. Intermediate C Programming. CRC Press. New York. Pp 9-27 (Chapter 4.3-)
- Tomasz Müldner. 2000. C for Java Programmers. Addison Wesley Longman. Reading, MA. 499pp. Chapter 8. (available in the Engg/CS Library)

6

## The Lesson of swap ( )

- Through pointers, a function can access the values of variables in another frame.
- But you already knew that
  - this is what happens in `scanf ( )` too!
- But only if the pointer references a frame below the current frame.

```
int m = 0;
return &m
```

These are  
not  
shoulders



7

## Using Pointers

1. Don't confuse `pj=pi` with `*pj=*pi`:
 

```
int i, j, *pi, *pj;
pi = &i;
pj = &j;
```
2. Should this work?
 

```
int i, j;
int *pi = &i;
int *pj = &j;
scanf("%d %d", pi, pj);
```
3. Why bother using `i` and `j`?
 

```
int *pi;
int *pj;
scanf("%d %d", pi, pj);
```

8

## const Keyword

```
const int *p;
```

- pointer to a constant integer, the value of `p` may change, but the value of `*p` can not

```
int *const p;
```

- constant pointer to integer; the value of `*p` can change, but the value of `p` can not

```
const int *const p;
```

- constant pointer to constant integer.

9

## Generic Pointers

- A pointer is just an address variable
- Regardless of what it points to, a pointer is just an address variable
- What if it's **just** an address variable?
  - `void* pv;`
  - defines a generic, or typeless pointer `p`.
  - often cast to `(T*)p`
  - Generic pointers cannot be dereferenced
    - what would you dereference it to?
  - **Must** cast: `(double*)pv`

10

## Everything About Nothing

- Special “zero” value that is useful to initialize pointers, and then to compare pointer’s value:  

```
if(p == NULL) {}
```
- NULL defined in six headers, including
  - `stdio.h`
  - `stdlib.h`

11

## Pass by Reference

- Consider:  

```
void decompose(double x, long *int_part, double *frac_part){  
    *int_part = (long) x;  
    *frac_part = x - *int_part;  
}
```
- A call of decompose:  

```
decompose(3.24159, &i, &d);
```

12

## One-Dimensional Arrays in C - I

### C arrays:

- have a lower bound equal to zero
- ANSI C: arrays are static - their size must be known at compile time (can be defined at run time in C99).

### To define an array:

```
type arrayName[size];
```

- For example

```
1. int id[1000];
2. #define SIZE 10
   double scores[SIZE+1];
```

13

## One-Dimensional Arrays in C - II

- It is good programming practice to define size of an array with a macro

```
#define N 100
...
int a[N];
for (i=0, sum=0; i < N; i++)
    sum += a[i];
```

- Avoids error of falling off end of array

14

## Array Compile-time Initialization

- Most common form:  

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```
- If list of initializers is shorter, remaining elements initialized to zero.  

```
int a[10] = {1, 2, 3, 4, 5, 6};
```
- Can omit length of array when initializing  

```
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

15

## Array Subscripting

- `a[expr]`, when `expr` is an integer expression
- `a[expr]` can be used as with ordinary variables
  - `a[0] = 1;`
  - `printf("%d\n", a[5]);`
- Idioms with for loops  

```
for (i=0; i < N; i++)
    a[i] = 0;           /* clears a */
for (i=0; i < N; i++)
    scanf("%d", &a[i]); /* reads data into a */
```

16



## Arrays and Pointers (I)

- When an array is declared, the compiler allocates enough contiguous space in memory to contain all the elements of array.
- **The array does not know how big it is. That's your job.**
  - But what about `sizeof()`?
    - See later
- A single-dimensional array is a typed constant pointer initialized to point to a block of memory that can hold a number of objects.
 

```
int id[10];
int *pid;

pid = id;
id = pid; //FAIL, but why?
```

17

## Arrays and Pointers (II)

- Can access first element of an array using:
 

```
int a[10];
a[0] = 5;
```
- or through a pointer:
 

```
int *p;
p = &a[9];
*p = 5;
```

18

## Pointer arithmetic

- WAT?
- What it is and how it works (mentioned before)
 

```
int a[10];
int* pa;
// Demonstration only. Do not try this at home.
for(pa = a; pa<a+10;pa++){
    *pa = 0;
}
```
- Summary: *"You could do that"*
- Likely you'll see it.

19

## Arrays and Functions

- Length of array is left unspecified:
 

```
int f(int a[]){...}
```
- If length is needed, must be specified with extra parameter:
 

```
int f(int a[], int n){...}
```
- `bigInt.c`
  1. Passing an array to a function
  2. Processing an array
  3. Why `sizeof()` is deceptive

20

```
int bigInt(int arr[], int iNarr){
    int i;
    int big = arr[0];
    printf("bigInt: sizeof(a): %lu bytes\n", sizeof(arr));
    for(i=1; i<iNarr; i++){
        if(arr[i] > big)
            big = arr[i];
    }
    return big;
}
```