ECE 220: Computer Systems & Programming

Lecture 10: Strings and Multi-dimensional Arrays

Relationship between Pointers and Arrays

- An array name points to the first element in the array.

```
char word[10];
char *cptr;

cptr = word; // points to word[0]
```

- What is difference?

cptr is a variable, but the name "word" is not a variable.

```
cptr = cptr + 1;
word = word + 1; // compile error
```

Correspondence between Pointer and Array Notation

```
char word[10];
char *cptr;

cptr = word; // points to word[0]
```

- Each line below gives three equivalent expressions:

`cptr`	`word`	`&word[0]`
`(cptr + n)`	`word + n`	`&word[n]`
`*cptr`	`*word`	`word[0]`
`*(cptr + n)`	`*(word + n)`	`word[n]`

Passing Array as Arguments

- C passes arrays by reference
  - The address of the array (*address of the first element*) is written to the function's activation record.

```
#define MAX_NUM 10
int main(){
    int array[MAX_NUM];
    int result;
    result = func(array);
}
```

Which function declaration is correct?

```
int func(int array[]);

int func(int *array);

int func(int array[MAX_NUM]);
```

Example: Roll a dice!

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
int main(){
    srand(time(0));
    int rnd_number[3];

    // update rnd_number as 3 integers
    roll_a_dice1(&rnd_number[0],&rnd_number[1], &rnd_number[2]);


    // update rnd_number as array
    roll_a_dice2(rnd_number);
```

```
// generate 3 random numbers, 1-6
// rand(): range between 0 to INT_MAX

void roll_a_dice1(int* one, int* two, int* three){
    *one = (rand () % 6) + 1;
    *two = (rand () % 6) + 1;
    *three = (rand () % 6) + 1;
}

void roll_a_dice2(int array[]) {
//void roll_a_dice2(int *array)
//void roll_a_dice2(int array[3])
    ??? = (rand () % 6) + 1;
    ??? = (rand () % 6) + 1;
    ??? = (rand () % 6) + 1;
}
```

```
void roll_a_dice1(int* one, int* two, int* three){
    *one = (rand () % 6) + 1;
    *two = (rand () % 6) + 1;
    *three = (rand () % 6) + 1;
}

void roll_a_dice2(int array[]) {
//void roll_a_dice2(int *array)
//void roll_a_dice2(int array[3])
    array[0] = (rand () % 6) + 1;
    array[1] = (rand () % 6) + 1;
    array[2] = (rand () % 6) + 1;
}
```

```c
void roll_a_dice1(int* one, int* two, int* three){
    *one = (rand () % 6) + 1;
    *two = (rand () % 6) + 1;
    *three = (rand () % 6) + 1;
}

void roll_a_dice2(int array[]) {
//void roll_a_dice2(int *array)
//void roll_a_dice2(int array[3])
    *(array) = (rand () % 6) + 1;
    *(array+1) = (rand () % 6) + 1;
    *(array+2) = (rand () % 6) + 1;
}
```

Exercise: Implement a function to reverse an array

```
/*array_reverse(): reverses an integer array, such that the first
element will become the last element, the second element will
become the second to last element and so on. This function takes
two arguments: a pointer to an integer array and its size.*/

void array_reverse(int array[], int n)
{

}


void print_array(int array[], int n)
{

}
```

Exercise: Implement a function to reverse an array

```
void array_reverse(int array[], int n)
{
int i;
int temp;
for(i = 0; i < n/2 ; i++)
{temp = array[i];
array[i] = array[n-1-i];
array[n-1-i] = temp;
}
void print_array(int array[], int n){
int i;
for(i=0; i<n; i++){
printf("%d ", *(array+i) ); //printf("%d ", array[i]);
}
printf("\n");
}
```

Strings

Strings are sequence of chars that represent text. They are declared as arrays of type char.

- Allocate space for a string just like any other **array**

```
char buf[200];
```

- Space for string must contain room for terminating zero.
- Null terminating strings-
- A null-terminated string is a sequence of characters stored in memory, typically as an array, that is marked with a special null character (i.e. a special character sequence , '**\0**', used in C and C++, and having an integer value of 0) at its end. This null character signifies the termination point of the string.

- **Special syntax for initializing a string:**

```
char buf[200] = "abc";
buf[0] = 'a';
buf[1] = 'b';
buf[2] = 'c';
buf[3] = '\0';
```

How about this?

```
char buf[200];
buf = "abc";
```

Exercise: Copy a string from source to destination

```
int main(){

    char buf[200];

    //buf = "ABC"; // compile error
    string_copy(buf,"ABC");
}
```

Exercise: Copy a string from source to destination

```c
#include <stdio.h>
#include <string.h>
void string_copy(char des[], char src[]);
int main(){
    char des[10];
    string_copy(des, "ABC");
    printf("%s\n", des);

    strcpy(des, "ABCDE");// from string.h
    printf("%s\n", des);
return 0;
}
```

```c
void string_copy(char des[], char src[]){

    int i = 0 ;
    while(src[i] != '\0'){
        des[i] = src[i];
        i++;
    }
    des[i] = '\0';
}
```

I/O with Strings

printf and scanf use “%s” format character for string.

`printf`: print characters up to terminating zero

```
char buf[10]="abc";
printf("%s\n", buf);
```

`scanf`: read characters until whitespace, store result in string, and terminating with zero.

```
char input[10];
scanf("%s", input);
```

How about this?

```
char input[10];
scanf("%s", &input[0]);
```

Q1.

```
char temp[100] = "abcdef";
temp[3] = '\0';
printf("%s", temp);
```

Q2.

```
char temp[100] = "abcdef";
printf("%s", &temp[2]);
```

Read Strings

- fgets vs scanf

```
char buf[SIZE_BUF];
fgets(buf, SIZE_BUF, stdin);
scanf("%s",buf);
```

stdin→ keyboard
stdout→ monitor

fgets: Read characters from stdin(keyboard)
and store them into buf as a string
until SIZE_BUF-1 characters
or a newline or the end-of-file (EOF)

scanf: Read characters from stdin(keyboard)
and store them into buf
until whitespace characters

Read Strings

- `sscanf:` Read formatted data from string
  and retrun the number of items successfully read

```
char buf[SIZE_BUF]=“abc1 34 21”;
int rc;
int num1, num2;
char str[20];
rc = sscanf(buf,"%s%d%d", str, &num1, &num2);
```

```
rc=3, str=abc1, num1=34, num2=21
```

Multi-dimensional Arrays

```
int a[4];
```

	col0	col1	col2	col3
row0	a[0]	a[1]	a[2]	a[3]

```
int a[2][3];
```

	col0	col1	col2
row0	a[0][0]	a[0][1]	a[0][2]
row1	a[1][0]	a[1][1]	a[1][2]

In memory,

	offset
	offset
a[0]	0
a[1]	1
a[2]	2
a[3]	3

offset=
(row index) * (size of column) + column index

	offset
a[0][0]	0
a[0][1]	1
a[0][2]	2
a[1][0]	3
a[1][1]	4
a[1][2]	5

Multi-dimensional array is stored in row-major order

Passing Array as Arguments

- C passes arrays by reference
  - The address of the array (*address of the first element*) is written to the function's activation record.

Must specify the **second** dimension

1D array

```
void func(int a[6]);
void func(int a[]);
void func(int *a);
```

2D array

```
void func(int a[2][3]);
void func(int a[][3]);
```

```c
#define ROW 2
#define COL 3

void print_2D(int a[][COL]){
    int i, j;
    for(i=0; i<ROW; i++){
        for(j=0; j<COL; j++){
            printf("%d ", a[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}


void print_2D_ptr(int *a){
    int i, j;
    for(i=0; i<ROW; i++){
        for(j=0; j<COL; j++){
            printf("%d ", a[i*COL+ j]);
        }
        printf("\n");
    }
    printf("\n");
}
```

compiler needs to know the size of column to calculate the memory offset

```c
int main()
{
    int two[2][3] = {{1,2,3},{4,5,6}};
    print_2D(two);
    print_2D_ptr(&two[0][0]);
}
```

pass the array as a pointer

lose the 2D-array shape information

offset=
(row index) * (size of column) + column index

Initialize Multi-dimensional Array

```
int a[2] = {1,2};
int a[] = {1,2};


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

Compile error!

Compile error!

Exercise: Exchange two rows in matrix

```
// Swap x-th row and y-th row
void matrix_change(int x, int y, int src[N][M])
{

}



// Transpose src matrix and store in des matrix
void matrix_tr(int src[N][M], int des[M][N])
{

}
```