Introduction to C

Tutorial 10: Pointers, Arrays, and Strings
Last week…

- Pointers
- Call by reference
Agenda

• Relationship between pointers and arrays (pointer arithmetic)
• Passing arrays to functions
• Strings
• Example
Pointer-Array Relationship
Review: Memory snapshot of an array

• Example

```c
int a[10] = {5, 8, 10, 0, 0, 0, 0, 0, 0, 0};
```

• What is the value of each expression?

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The name of the array is almost a pointer

• The name of an array without the index is the same as the address of the first location in the array.
  \[
  \&a[0] \quad \text{equals to} \quad a
  \]

• You can use the * operator on this address.
  \[
  a[0] \quad \text{equals to} \quad *\&a[0] \quad \text{equals to} \quad *a
  \]

• Conclusion: the name/variable of the array works almost like a pointer.

You cannot change the address of the array in memory!
• Addition and subtraction (\(--\, ++\, ,-\, ,+)\) are defined for pointers.

• What will the value of \(p\) be after the following?

\[
\begin{align*}
\text{int } num = 52; \\
\text{int } *p; \\
p = &\text{ num}; \\
p = p + 1;
\end{align*}
\]

3188 \hspace{1cm} 52

• The definition is based on the size of the type being pointed to.

– It’s like pointing to the next variable in memory

\[
\begin{align*}
p = p + k; \\
\text{new } \text{addr.} = \text{old } \text{addr.} + k * \text{sizeof}(\ast p)
\end{align*}
\]
• Adding \( k \) to an address is like advancing \( k \) variables ahead for the same type.

\[
\begin{align*}
\&a[k] & \quad \text{equals to} \quad a+k \\
\&a[k] & \quad \text{equals to} \quad *\&a[k] \\
a[k] & \quad \text{equals to} \quad * (a+k)
\end{align*}
\]
• If a pointer p is pointing to the start of the array a, you can access element i with p[i].

```c
int a[3];
int *p;
p = a;
```

<table>
<thead>
<tr>
<th>&amp;a[k]</th>
<th>equals to</th>
<th>a+k</th>
<th>equals to</th>
<th>p+k</th>
<th>equals to</th>
<th>&amp;p[k]</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[k]</td>
<td>equals to</td>
<td>*(a+k)</td>
<td>equals to</td>
<td>*(p+k)</td>
<td>equals to</td>
<td>p[k]</td>
</tr>
</tbody>
</table>
Examples

• What is the difference between the two code snippets?

```c
int a[3] = {5,8,10};
int *p;
p = a; /* or p = &a[0]*/
p = p + 1;
```

```c
int a[3] = {5,8,10};
int *p;
a = a + 1;
p = a; /* or p = &a[0]*/
```

• What would the following code print for the variables declared above?

```c
printf("%d\n", sizeof(a) );
printf("%d\n", sizeof(p) );
printf("%d\n", sizeof(*p) );
```
Passing arrays into functions
Passing a one dimensional array to a function

- Pass the address of the first element to the function.
- Example

```c
void f1(int *a);
void f1(int a[]);
void f1(int a[25]);
```

Example of a call:
```c
int a[10];
f1(a);
f1(&a[0]);
```

To avoid violating the bounds of the array, we must also pass in the length of the array into the function.

The compiler ignores all values written in the braces.
Passing in the length

- Method 1: Pass in the length as another parameter

```
#include <stdio.h>

#define N 20

void print_vector(int *arr)
{
    int i;
    for (i=0; i < N; i++)
        printf("%d ",arr[i]);
}
```

- Method 2: Establish the length of the array as a constant using a #define

```
#include <stdio.h>

#define N 20

void print_vector(int *arr)
{
    int i;
    for (i=0; i < N; i++)
        printf("%d ",arr[i]);
}
```
## Comparison between methods

<table>
<thead>
<tr>
<th>Method 1</th>
<th>Method 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of the array passed as parameter.</td>
<td>Size of the array defined as a <strong>constant.</strong></td>
</tr>
<tr>
<td><strong>Can not</strong> create static helper array of the same size in function</td>
<td><strong>Can</strong> create helper array of the same size in function</td>
</tr>
<tr>
<td>Can work with arrays of any size</td>
<td>Only work with arrays of same size as constant</td>
</tr>
</tbody>
</table>
Finding a maximum

- This function receives an array and its size as parameters, and returns the maximum element in the array.

```c
int find_max(int a[], int n)
{
    int i;
    int max = a[0];
    for (i=1; i<n; i++) {
        if (a[i] > max) {
            max = a[i];
        }
    }
    return max;
}
```

Why can’t you initialize max to 0?
• Given the function from the previous slide, what will the program print?

```c
int main(void) {
    int data[] = {6, 9, 7, 1, 3};
    int max1;
    int max2;

    max1 = find_max(data, 5);
    max2 = find_max(data+2, 3);

    printf("%d\n", max1);
    printf("%d\n", max2);
    return 0;
}
```
Two dimensional array as a parameter (1)

- If you want to write a function that assumes a fixed row length, the matrix can be passed in the following manner:

```c
void f(int a[][4], int rows)
```

- The `rows` variable is passed so that the boundaries are not exceeded.
- This function will assume that `a` is a matrix, so the element `i,j` can be accessed with:
  - where `0 ≤ i ≤ rows-1, 0 ≤ j ≤ 3`
- The function call will look like:

```c
int m[9][4];
f(m, 9);
```
Two dimensional array as a parameter (2)

• But if you want to write a function that works for a matrix of arbitrary size, you must pass a pointer to the first element, plus the height and width of the matrix.

```c
void f(int *p, int rows, int cols)
```

• Now, from the perspective of the function, p is no longer a matrix, but instead it is a pointer.

• Therefore in the function you cannot use p[i][j], but instead, using i,j:

```c
*(p + i*cols + j)
```

– $0 \leq i \leq \text{rows}-1$, $0 \leq j \leq \text{cols}-1$

• The function call will appear as:

```c
int m[9][4];
f(m, 9, 4);
```
Strings
What is a string?

• A string is a series of characters (of type char) that ends with a '\0' character
  – This special character '\0' marks the end of a string.
  – The ASCII value of this character is 0. This is different than the character ‘0’, whose ASCII value is 48.
• Thus, a string is actually an array of characters that ends with '\0'
• Example of a string:
  char s[ ] = {'H', 'e', 'l', 'l', 'o', '!', '\0'};
• You can also declare the array in an easier way.
  char s[ ] = "Hello!";
• Strings in C are called “null terminated” strings because they use the 0 value to mark the end of the string.
C support for strings

• Inputting a string with %s in scanf.
  ```c
  char str[10];
  scanf("%s", str);
  ```
  This provides a max of 9 characters of input, since ‘/0’ takes up a spot!

• Printing a string with %s in printf
  ```c
  char str[10] = "hello";
  printf("%s", str);
  ```

• Using a fixed string
  ```c
  int age;
  scanf("%d", &age);
  printf("Do you have future? %s", (age<30)?"yes":"no");
  ```

string.h is the library that deals with strings
What is the length?

• What is the length of the string, and of the array?

```c
char str1[] = "My Cat";
char str2[10] = "My Cat";
char str3[] = {'M','y',' ','C','a','t'};
char str4[7] = {'M','y',' ','C','a','t'};
char str5[7] = {'M','y','\0','C','a','t'};
char str6[7] = {'M','y','\0','C','a','t'};
char str7[] = "";
```
Passing a string to a function

• Similar to arrays, you can pass a pointer to the first character in the string to a function.
• Since the last character is ‘\0’, there is no need to pass the length to the function.
• What will the following code print?

```c
#include <stdio.h>

void print_string(char *s)
{
    printf("%s\n", s);
}

int main()
{
    char str[] = "Hello World";
    print_string(str);
    print_string(str+6);
    return 0;
}
```
Exercise

• Write a function, that when given two strings s1 and s2, checks if s1 ends with the substring, s2.
  – If yes, return 1
  – If not, return 0
• For example, for s1=“Hello World!” and s2=“World!” the function will return 1.
• For s1=“Hello World!” and s2=“Hello”, returns 0
• For s1=“World!” and s2=“Hello World!” returns 0

Where is this location in s1?

Size of s1 must be greater than size of s2.
```c
#include <string.h>

int is_ending_substring(char *s1, char *s2) {
    int len1 = strlen(s1);
    int len2 = strlen(s2);

    if (len2 > len1) { /* s2 is definitely not a substring */
        return 0;
    }

    if (!strcmp(s1+len1-len2, s2)) {
        return 1;
    }
    return 0;
}
```

equals to
\&s1[len1-len2]
Exercise: Strings

• Write a function that gets two strings, s1 and s2, and returns the number of times that s1 appears in s2.
• The function should be case-insensitive, meaning it ignores whether letters are uppercase or lowercase.
• Both strings should not change
• For example, for:

  s1 = aba
  s2 = kaba 2 abababaabaBA$

  The result in this case is 5.
The way to the solution

You cannot use `strncmp` because:
- `strncmp` does not compare substrings inside of other strings
- `strncmp` is case sensitive

We’ll write a special function that takes care of these cases
- To work with a substring in the middle of a word, we’ll pass an `int n` which signifies the length of the string to compare.
- To make our function case-insensitive, we’ll write a function that compares characters and ignores the case.
Solution: Part A

```c
#include <string.h>

char upCase(char c)
{
    if (c >= 'a' && c <= 'z') {
        return c - 'a' + 'A';
    }
    return c;
}

int isEqualStringN(char s1[], char s2[], int n)
{
    int i;
    for (i=0; i<n; i++) {
        if (upCase(s1[i]) != upCase(s2[i])) {
            return 0;
        }
    }
    return 1;
}
```

Translates lowercase to uppercase. Every uppercase letter stays the same.

Case-insensitive comparison of the first n letters of two strings.
int CountSubstring(char *s1, char *s2)
{
    int len1 = strlen(s1), len2 = strlen(s2);
    int i, count = 0;
    for (i = 0; i <= len2-len1; i++) {
        if (isEqualStringN(s1, s2+i, len1)) {
            count++;
        }
    }
    return count;
}