CMSC 313
Introduction to Computer Systems
Lecture 5
Arrays and Strings, cont. & Structures and Unions

Alan Sussman
als@cs.umd.edu

Administrivia

- Project 1 questions?
  - due Tuesday, 9/22

Parts of Chapters 8 and 9, Reek

ARRAYS AND STRINGS

Use of symbolic constants

- #define preprocessor directive
  `#define name value`
- All occurrences of **`name`** in the source file are replaced by **`value`**
- Can use this to define constants for things such as array sizes and other values to improve program maintainability

```
#define ARR_SIZE 3
int main() {
  int i, a[ARR_SIZE] = {1, 2, 3};
  multiply_array(5, a, ARR_SIZE);
  printf("[%d", a[0]);
  for (i = 1; i < ARR_SIZE; i++)
    printf(", %d", a[i]);
  printf("]\n");
  return 0;
}
```

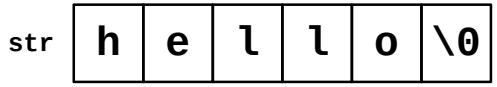
Strings in C

- There is no String type in C
- In C, a string is defined as a sequence of characters that is followed by a byte with the value zero
  - often called: "zero byte", "null byte", "NUL"
  - represented as the character literal **`'\0'`**
  - "null byte" is NOT the same thing as "null pointer"
- Since arrays are contiguous in memory, and **`char`**s are all one byte in size, we can use arrays of **`char`**s to hold strings
- **`printf()`** format specifier for strings is **`%s`**

String initialization

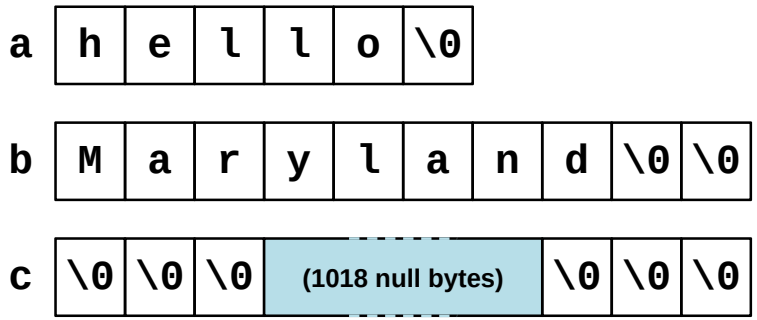
- Because character arrays are so closely related to strings, they can be initialized with string literals as well as standard array initializers
- But don't forget that the null byte needs to be stored as well
- Example:

```
char str[6] = "hello";
```



More examples of string initialization

```
char a[] = "hello";
char b[10] = "Maryland";
char c[1024] = "";
```



Basic string library functions

- C has many different functions for working with strings; to use these, you must **`#include <string.h>`**
- We're only covering a small subset here; if you ever want to see all of them, more information can be found in the string.h man page
  - Note: the prototypes there are slightly different than what we'll be covering here, because we haven't covered pointers yet, but functionality is the same

String library functions, cont.

- String length:

  `size_t strlen(char str[]);`

  – returns the length of the string pointed to by the string passed in as a parameter

  – string length is the number of characters in the string, **not counting** the null byte

  – Example:

```
char str[] = "ice cream";
printf("\"%s\": %d chars\n", str, strlen(str));
```

Output:

```
"ice cream": 9 chars
```


A possible `strlen()` implementation

```
size_t strlen(char str[]) {
  size t i;
  for (i = 0; str[i]; i++)
    ;
  return i;
}
```

- The integer type **`size_t`** is discussed in the project #1 handout
- What would happen if you passed an uninitialized character array into this function?

String library functions, cont.

- Comparing strings:

  `int strcmp(char s1[], char s2[]);`

  – works just the same as **`s1.compareTo(s2)`** did in Java:

  - returns negative number if **`s1`** is less than **`s2`**
  - returns positive number if **`s1`** is greater than **`s2`**
  - returns 0 if **`s1`** and **`s2`** match character for character

  – Example:

```
if (strcmp(str1, "hello") == 0)
  printf("str1 is \"hello\"\n");
```


A possible `strcmp()` implementation

```
int strcmp(char s1[], char s2[]) {
  int i;
  for (i = 0; s1[i] && s2[i]; i++)
    if (s1[i] != s2[i])
      break;
  return s1[i] - s2[i];
}
```

- Notice the return statement subtracts characters; remember that **`char`** is an integer type

String library functions, cont.

- Copying strings:

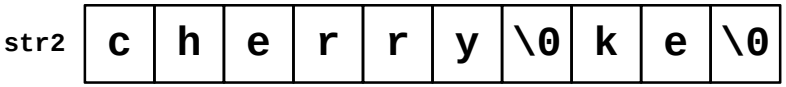
  `strcpy(char dest[], char src[]);`

  – copies the string in **`src`** to **`dest`**

  – it is up to the programmer to ensure that dest is an array with enough characters to hold the string
    - being lazy with this function can result in buffer overflows

  – Example:

```
char str[] = "cherry";
char str2[10] = "milkshake";
strcpy(str2, str);
```



A possible `strcpy()` implementation

```
void strcpy(char dst[], char src[]) {
  int i = 0;
  while (src[i]) {
    dst[i] = src[i];
    i++;
  }
  dst[i] = '\0';
}
```

- What expression gives the minimum size of the array **`dst`** (to ensure safe execution)?

Chapter 10, Reek

STRUCTURES AND UNIONS

Structures

- Like arrays, hold multiple items
- Items need not be of the same type
- Items referred to by field names, not numerical indices
- You can assign the value of another structure to a structure
- You cannot use **`==`** or **`!=`**
- Similar to a Java class with all public fields and no methods

Creating structures

- Example:

```
struct employee {
  int id_number;
  char last_name[10];
  char first_name[10];
  double salary;
} emp1, emp2;
```

- Declares two variables (**`emp1`** and **`emp2`**) of type **`struct employee`**
- **`employee`** is called the tag of these two structs
  - used to differentiate between different kinds of structs

Structure declarations

- More formally, this is the syntax for declaring structures (or structure types):

  `struct` ***tag*** **`{`** ***member-list*** **`}`** ***variable-list*****`;`**
- Omitting *variable-list* creates a new type
- Omitting *member-list* (and **{}**) declares variables of an existing **`struct`** type
- Omitting *tag* means you create a unique type for the variables listed
  - even if *member-list*s are the same
  - prevents use of those **`struct`**s as function arguments

Accessing fields of a structure

- Dot operator:

```
struct point {
  int x, y;
};

struct point p1, p2, points[5];
p1.x = 17;
p2.y = 22;
points[0].x = 13;
points[0].x++;
```


The **`typedef`** keyword

- You can give types new names
  - eases readability and maintainability
- **`typedef`** ***existing-type new-name*****`;`**
  - the type may be created along with the **`typedef`** usage, as we'll see with structures
- **`typedef double Dollars;`**
  `Dollars x, y = 1.25;`
  - now you know that **`x`** and **`y`** shouldn't be assigned values like **`sqrt(15)`**
- Using caps to start **`typedef`**'d names helps set them apart from other types

Combining `typedef` and `struct`

- Combining the two keywords:

```
typedef struct {
  int i;
  char ch;
} Ex_struct;
...
Ex_struct a[10], b;
```

- Structure definitions (either form) should be placed in header files if the structures are used across multiple files