CMSC 313
Introduction to Computer Systems
Lecture 5
Arrays and Strings, cont. & Structures and Unions
Alan Sussman
als@cs.umd.edu

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

```c
#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 *chars* are all one byte in size, we can use arrays of *chars* 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:
  ```c
  char str[6] = "hello";
  ```

More examples of string initialization

```c
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:

```c
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:
  ```c
  char str[] = "ice cream";
  printf("%s": %d chars
", str, strlen(str));
  Output:
  "ice cream": 9 chars
  ```

String library functions, cont.

• Comparing strings:

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

A possible `strlen()` implementation

```c
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?

A possible `strcmp()` implementation

```c
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:

\[
\text{strcpy}(\text{char dest}[\text{ ], char src[ ]});
\]

– 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:

\[
\begin{align*}
\text{char str[ ]} & = \text{"cherry"}; \\
\text{char str2[10]} & = \text{"milkshake"}; \\
\text{strcpy(str2, str)};
\end{align*}
\]

\[
\begin{array}{c}
\text{str2} \\
\begin{array}{c}
\text{c}
\text{h}
\text{e}
\text{r}
\text{r}
\text{y} \\
\text{\0}
\text{k}
\text{e} \\
\text{\0}
\end{array}
\end{array}
\]

A possible \text{strcpy()} implementation

\[
\begin{align*}
\text{void strcpy(char dst[ ], char src[ ])} & \{ \\
\text{int i } & = \text{0}; \\
\text{while (src[i]) } & \{ \\
\text{dst[i]} & = \text{src[i]}; \\
\text{i } & += \text{1}; \\
\} \\
\text{dst[i]} & = \text{\'\0;} \\
\}
\end{align*}
\]

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

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-lists are the same
  – prevents use of those structs 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:
  ```c
  typedef struct {
    int i;
    char ch;
  } Ex_struct;
  ```

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

  ```c
  Ex_struct a[10], b;
  ```