

Strings in C

CSE 2421

Strings in C – Intro

- ▶ C does not have a string data type.
- ▶ In C, strings are stored as sequences of characters in memory (i.e. a character array). The individual characters will ***always*** be stored in contiguous bytes in memory.

Strings in C – Intro cont.

- ▶ There are two different kinds of data “strings” in C:
 1. Strings which are stored as string literals (“read only” strings).
 2. Strings which are arrays of characters (read–write strings).
- ▶ We will look at each of these kinds of strings.
- ▶ Pointers are also closely related to strings in C, and whenever we use a pointer to a string, we must be careful to distinguish between the two types of strings above.

Example declarations

```
char *string1 = "Go Bucks!";
```

```
char string2[10] = "Go Bucks!";
```

- ▶ string1:

- The individual chars in string1 make up a **read-only string literal** (a *string constant*).
- string1 (the identifier) is a pointer to char, and it is *a variable* (it can be made to point to a different string).

- ▶ string2:

- The individual chars in string2 are *variables* (we can change the chars in the string, so the elements in string2 can be treated as they are in other arrays).
- The initial values of the 10 changeable chars are copied from the read-only string literal on the right side of the = sign
- string2 (the identifier) is *a constant* pointer to char (just as the name of any static array in C is normally a constant pointer to the first element).

In either case, the values stored in each of the individual characters would be the ASCII value of each alphanumeric or symbol and will end with a NULL value at the end of the string.

Read Only versus Read-Write strings

- ▶ `string1` on the preceding slide is a **read-only string** (also called a **string literal**), because the chars in the string will be stored in read-only memory.
- ▶ `string2` on the preceding slide is a **read-write string**, because it is stored in “ordinary” memory, which can be both read and written. (They are initialized from a read-only string literal)

Examples

- ▶ Consider this code:

```
char *string1 = "warehouse";
```

```
char string2[] = "container";
```

```
string1[6] = 'u';           /* Invalid - chars in a string literal, i.e.,
                             a read-only string, cannot be changed -
                             segmentation fault occurs */
```

```
string1 = "warehouse"; /* Valid - a pointer to char can be
made to point to a different string; the
```

string which string1 points to is still a read-only string, however */

[illegible]

String Basics

- ▶ A string is a sequence of zero or more characters which should be terminated by a null character (NUL byte), ‘\0’; the ASCII character code for ‘\0’ has the value 0. Thus, no string can contain this character *except at its end*.
- ▶ The null character termination is one of the fundamental differences between arrays of characters and other arrays.

String Basics cont.

- ▶ The null character termination is important, because library functions such as `printf()` and `scanf()`, among many others, use it to mark and determine where the string ends (i.e., the length of the string).
- ▶ Because of the null character termination, we do not have to pass the size or length of the string to functions which manipulate strings.
- ▶ If the null character termination is not present, various library functions which can manipulate strings may not behave correctly, because they will not be able to determine where the string ends. In such cases, if you are lucky you will get segmentation faults when your code runs.
- ▶ When you declare read only or read–write strings using “. . .”, (double quotes) the compiler stores ASCII representations of each character in your string and will add the null character termination at the end.

Null Character Termination Example

- ▶ Example:

```
int main () {  
    char string[] = {'e', 'x', 'a', 'm', '\0', 'p', 'l', 'e'};  
    printf("%s", string);  
    return (0);  
}
```

What will be printed?

Null Character Termination Example

- ▶ Example:

```
int main () {  
    char string[] = {'e', 'x', 'a', 'm', '\0', 'p', 'l', 'e'};  
    printf("%s", string);  
    return 0;  
}
```

What will be printed?

exam

Null Character Termination Example

- ▶ The string appears to consist of 8 characters, but because `printf()` treats the null character, `'\0'`, as the end of the string, the final three characters that were stored in the char array when it was initialized are not printed by `printf`.
- ▶ The **array** holds 8 characters. The string currently in the array needs 5; $4 + 1$ for the null

String Basics cont.

- ▶ The header file `string.h` contains prototypes and declarations needed to use string functions.
- ▶ The `-Wimplicit-function-declaration` flag **will** generate warnings if you do not include `string.h`, even though the code will compile
- ▶ `strcpy` for example returns a `char *` and not an `int`, so the default return type is wrong

String Length and '\0'

- ▶ The length of a string is the number of characters it contains before the terminating null character, if one is present (it always should be).
- ▶ Thus, when you use an array of characters as a string, you must make the size of the array **one greater than the string length**, or the maximum number of characters which the string can contain before the null character termination. Example:

```
char label[10] = {'c', 'o', 'n', 't', 'a', 'i', 'n', 'e', 'r'};
```

```
/*a string stored in label can have a maximum length of 9, so that  
there is space for the null character termination*/
```

Note: In this case, the compiler will **still** add the terminating '\0' at the end of the string, as long as you leave space for it).

Another Example

- ▶ The following two declarations are equivalent:
`char label[10] = {'c', 'o', 'n', 't', 'a', 'i', 'n', 'e', 'r'};`
`char label[10] = "container"; /* More usual declaration */`
- ▶ In both cases, the compiler will add the terminating null character when it stores the characters of the string in memory (as long as the length of the string is sufficient).
- ▶ Also notice that a character enclosed in single quotation marks is treated as **char**, whereas zero or more characters enclosed in double quotation marks are treated as a **string** (a sequence of contiguous chars).
- ▶ Look back at the slides on I/O in C to see how to read strings from input with `scanf()`.

Dealing with Individual Chars in Strings

- ▶ Since strings are stored as arrays in memory, individual chars can be accessed using an index:

```
char *string1 = "quit";  
char string2[12] = "slop";  
if (string1[0] == 'q') {  
    string2[1] = 't'; /* string2 changes from "slop" to "stop" */  
}
```

- ▶ Remember: Since string1 is a read-only string, although it is valid to read individual chars in the string, you cannot write to them. If you attempt to write individual char elements of a string which is in read-only memory, you will get segmentation faults.

Assignments with strings

- ▶ You can assign a read-write string to a `char *` type of string variable; in that case, the `char *` string is read-write (Remember, a `char *` is just a pointer, and in this case, points to a string in read-write RAM).

- ▶ Example:

```
char *string1;  
char string2[] = "Go Ducks!";  
string1 = string2;  
string1[3] = 'B';    /* No seg fault, because string1 */  
                    /* points to a read-write string */  
printf("string1 is: %s\n", string1);  
/*NOTE: not *string1 - WHY???? */
```


Assignments with strings – cont.

- ▶ HOWEVER, you cannot assign a `char *` type of string to a `char array` type of string.

```
char *string1 = "Go Bucks";  
char string2[11] = "Touchdown!";  
string2 = string1; /* INVALID – WHY?*/
```

The compiler will refuse to compile, and will output:

error: incompatible types when assigning to type 'char[9]' from type 'char *'

Pointer Arithmetic with Strings

- ▶ Pointer arithmetic may be used with strings which are arrays of char (and also with read-only strings which are char * and point to string literals, but only for reading):

```
char string1[8] = "quick";
```

```
char string2[5] = "dime";
```

```
if (*(string1 + 0) == 'q')
```

```
    *(string2 + 0) = 't';
```

```
printf("%s\n", string2);
```


time is printed

A note on printf and strings


- ▶ Normally, we do not pass *addresses* to printf, in order to print the value of a variable with a particular format specifier (%c, %i/%d, %f).
- ▶ With strings, however, it is different:

```
char *string1 = "Bucks";  
printf ("%s", string1);
```
- ▶ Since string1 is a **pointer**, printf gets the address where the first char in string1 is stored, and it will output a string of characters starting with the character at that address, until it encounters a null character.
- ▶ Notice that dereferencing of the pointer is **not used** here (this may seem surprising, but since we used a string format specifier (%s), printf “knows” that we want to print a sequence of chars, and *it expects a pointer to the first char*, so the dereference operator is not necessary, and will produce segmentation fault errors if used).
- ▶ If we wanted to print only a single char from string1, THEN we could (actually *must*) use dereference (or string1[0]): printf ("%c", *string1);


Character Operations

- ▶ The standard library has two types of character operation functions which operate on individual characters.
 - ▶ These characters can be contained in strings, and usually are.
 - ▶ The first type of function is used to classify characters according to certain characteristics, and the second type of function is used to transform them to a related character.
 - ▶ `ctype.h` is the header file where these functions are prototyped.
- 

Character Classification

- ▶ Each of these functions takes an integer argument which contains a ASCII character value.
 - ▶ All of these functions return an integer value which is used as a boolean.
 - ▶ They are listed on the next slide.
 - ▶ You can get a clue about what the function does from its name, but you can look up the details if you think one of these may be useful in some software you are writing.
- 

Character Classification Functions

- ▶ `isctrl()`
 - ▶ `isspace()`
 - ▶ `isdigit()` (also `isxdigit()` for hexadecimal digits)
 - ▶ `islower()`, `isupper()`
 - ▶ `isalpha()`
 - ▶ `isalnum()`
 - ▶ `ispunct()`
 - ▶ `isgraph()`
 - ▶ `isprint()`
- 

Character Modification Functions

- ▶ These functions are used to transform a character value, which is passed as an integer argument, to a related character value, which is returned as an integer.
- ▶ `int tolower(int ch);` `/*returns lower case if
ch is upper case */`
- ▶ `int toupper(int ch);` `/*returns upper case if
ch is lower case */`

String Library Functions in string.h

| | |
|-----------|---|
| strcat() | Appends a string |
| strchr() | Finds first occurrence of a given character |
| strcmp() | Compares two strings |
| strcmpi() | Compares two strings, non-case sensitive |
| strcpy() | Copies one string to another |
| strlen() | Finds length of a string (Note: uses '\0') |
| strlwr() | Converts a string to lowercase |

String Library Functions in string.h cont.

| | |
|-----------|---|
| strncat() | Appends n characters of string |
| strncmp() | Compares n characters of two strings |
| strncpy() | Copies n characters of one string to another |
| strnset() | Sets n characters of string to a given character |
| strrchr() | Finds last occurrence of given character in string |
| strrev() | Reverses string |
| strset() | Sets all characters of string to a given character |
| strspn() | Finds first substring from given character set in string |
| strstr() | Returns a pointer to the first occurrence of string s2 in string s1 |
| strupr() | Converts string to uppercase |

So what does this do again?

```
char *src, *dest;  
/* assume they get initialized */  
  
while(*dest++ = *src++);
```