C-Strings and String Functions

C-Strings and Functions

- Similar to other data types there are cases that you need to pass c-strings to a function, or to write a function that returns the address of a c-string.
- Since string name is a pointer, therefore a c-string can only appear as a pointer in a function parameter list. Example:

```
void fun (char *x);
```

  - The following function prototype is equivalent to above function prototype.

```
void fun (char x[]);
```

  - Remember again the size of x in these example prototypes is the size of a pointer (8 bytes on the computers in our ICT lab).

- A function also may return a char*:

```
char*  modify (char* x)
{
   // More code to modify the c-string pointed by x
   return x;
}
```

Warning: it is wrong to return a pointer that points to a local variable, because the lifetime of the local variable ends when function terminates.

Character Library Functions

- There are several C library functions that allow to identify the type of characters and their case.
- To use these functions, you should include <ctype.h> :
  - isdigit (ch) -- Returns true if ch is a digit (0-9)
  - islower(ch) -- Returns true if ch is a lower case letter (a-z)
  - ispunct(ch) -- Returns true if ch is a punctuation
  - iscntrl(ch) -- Returns true if ch is the control key
  - isspace(ch) -- Returns true if ch is the control key
  - toupper (ch) -- Returns uppercase ch
  - tolower(ch) -- Returns lowercase ch
  - isalnum(ch) -- Returns true if ch is alphanumeric character
- A Few Examples:

```
char mychar = 'b';
printf("%c", toupper(mychar)); // prints B
printf("%d", islower(mychar)); // prints 1 (true), as mychar holds a lower case char
printf("%d", isdigit(mychar)); // prints 0 (false)
```

Library Functions to Manipulate C-strings

- C doesn't support predefined type called `string` like in C++, or `String` in Processing. Therefore, you cannot use operators such =, +=, ==, >=, etc. to copy, concatenate, or compare c-stings.
- As stated earlier, a null-terminated array of characters represents as a c-string

```
char s1 [10] = "Apple";
char s2[ 10] = "Orange"
s1  =  s2;            // illegal statement
s1 += s2;             // illegal statement
If (s1 > s2) {  …}    // Compares address. Not a Lexicographic comparison
```

- There are several library function for string manipulation. To use these functions you need to include `<string.h>`. Some of the C-string library functions include:

strlen(s) -- Returns the length of a string. Examples:

```
char s[20] = "ABCD";
printf("%zu", strlen(s));        // prints 4
```

Note: size of s is 20 bytes but its string length is 4.

C-Strings – Library Functions

strcmp(s1, s2) - Compares s1 and s2: Returns zero if two strings are identical. Otherwise returns a positive integer if s1 is greater than s2, or a negative integer if s1 is less than s2.

```
char s1[20] = "BCC";
char s2[20]  = "BBC";
if (strcmp (s1,  s2) >  0)
      printf("%s is lexicographically greater than %s.", s1, s2);
```

strcpy(s1, s2) -- Copies s2 into s1:

```
char s1[20] = "ABCD";
char s2[20];
Strcpy(s2, s1);
printf("%s", s2);            // prints ABCD
```

strcat(s1, s2) -- Appends s2 to the end of s1.

```
char s1[20] = "ABCD";
char s2[20]  = "XY;
strcat(s2, s1);
printf("%s", s2);            // prints: XYABCD
```

Strings Functions that Return a char* Pointer

- Functions **strcpy** and **strcat** also return a char pointer (char*).
- The returned pointer points to the first argument of the function (in the following call to strcpy, s1), and can be used for different purposes:

```
char s1[5] = "Red";
char s2[5];
printf( "%s", strcpy(s1, s2));
```

- Or:

```
char s1[5] = "CM";
char s2[8] = "EN";
printf( "%s", strcat(s2, strcat(s1, "-339")));
```

  - First, function **strcat** appends string "-339" to the end of s1 ("CM") and returns "CM-339" to the outer call of **strcat** that receives s2 as its first argument. Then it concatenates string "CM-339" to the end of s2 (which is "EN").
  - Therefore the final output is: ENCM-339

Closer look at the **strcpy**

How strcpy works

- First lets take a look at a possible function-prototype for `strcpy`:

  char* strcpy(char *dest, const char* source);

- strcpy receives two char pointers as its arguments and returns a char pointer that points to same string that dest is pointing.
- Notice that **source** is a pointer to **const** but **dest** isn't. **Why?**
- The destination array must have sufficient space to hold the copy of source plus null-terminator. Otherwise will create an invalid string. Example:

```
char s1[10];
char s2[3];
strcpy(s1, "Orange");  // oK
strcpy(s3, "Orange");          // possible runtime ERROR
```

How `strcpy` works

Lets have the simple C program that uses `strcpy` and then take a look at the possible code for `strcpy`:

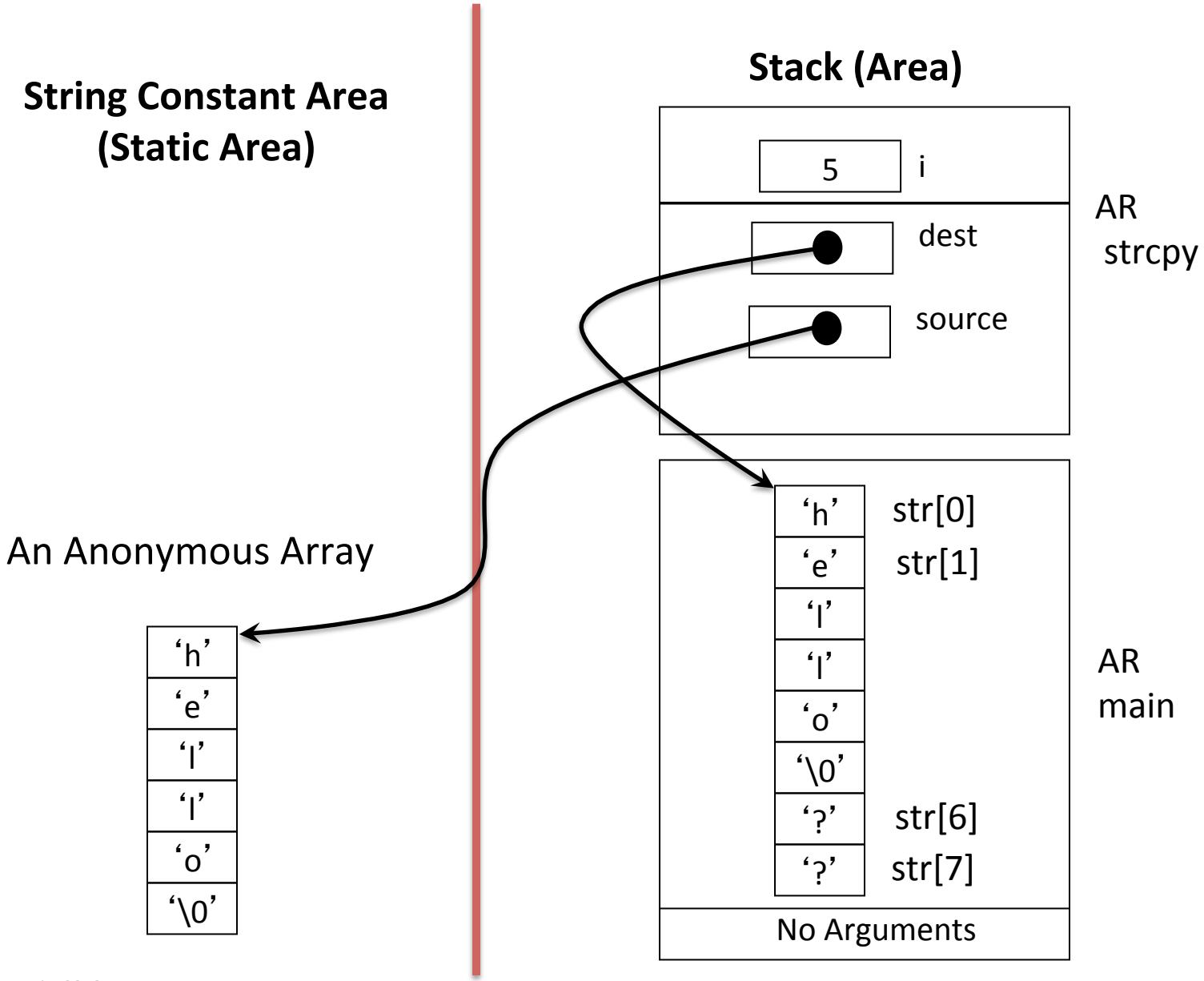
```
#include<string.h>
int main()
{
    char str[8];
    strcpy(str, “hello”);
    printf(“%s”,  str);
    return 0;
}
```

One Possible Definition for `Strcpy`

```
char* strcpy(char *dest, const char* source)
{
        int i=0;

        while (source[i] != '\0')
        {
            dest[i] = source [i];
             i++;
        }

        dest[i] = '\0';

         // POINT ONE

         return dest;
}
```

- Let's draw the AR diagram for point 1.

AR Diagram for Point-One in the strcpy Function (previous slide)



A Brief Note on Standard I/O Functions to Read Characters and Strings

Reading Strings and Characters using scanf (from one of the previous set of slides)

- scanf uses %s as a type identifier to read a string (up to a whitespace) from keyboard .

```
char lastName[25];
printf ("Enter your last name: ");
scanf ("%s", lastName);
```

  - Three character: spacebar, tab and return are considered as whitespace characters.
- scanf doesn't need an address operator to read a string. Why?
- scanf uses %c as a type identifier to read a character.

```
scanf ("%c", &lastName[0]);
```

  - Needs address operator to read a character.
- scanf is not the only library function to read strings and characters. There are other library functions such as:

  gets, fgets, getc, fgetc.

Reading characters

- Functions to read characters
  - int getchar(void);
    - Returns the character read, or EOF (-1), if reaches end of file or error.
    - Maybe implemented as a macro.
  - int fgetc(FILE *fp);
    - Returns the character read, or EOF (-1), if reaches end of file or error.

```
char s[5];
int c = fgetc(stdin);
if( c != EOF)
    s[0] = (char) c;
```

  - int getc(FILE *fp)
    - Similar to fgetc. Maybe implemented as a macro. Be aware of side effects of macro.
- Functions to read c-strings
  - gets (**not recommended** – can cause buffer overflow and potential security risk)
  - char *fgets (char* address, int n, FILE* fp);
    - Recommended as a better way to read string of character.
    - Has control over the size of user input. Reads a sequence of characters up to end of line character or maximum of n-1 characters.
    - Appends '\0'
    - Returns a pointer to the string buffer if anything was written to it, or a null pointer if an error occurred or if the file position indicator was at the end of the file
- Note: Before using any of the functions read chapter 17 in your textbook (C in a Nutshell)

Storage Classes in C

Overview of different memory segments

- There are different types of memory allocations for C/ C++ program's data:
  1. Stack-allocated memory or also know as automatic allocation.
  2. Static-allocated memory.
  3. Heap-allocated memory.
- We already discussed the first type when we introduced the activation records. The first and second types are allocated at the compilation time.
- The third type is allocated at the runtime. This one will be discussed in detail later in this course.

What is Static Allocation

Static Allocation

- Static allocation happens before main starts.
- Static variables are initialized to zero automatically. Or they can be initialized manually.
- Their lifetime is the life time of the program.
- Their scope depends on the type of the declaration.
- Major Examples of static allocation
  - String constants
  - Global variable
  - Local static variables
  - Others which won't be discussed in ENCM 339

Explicit Allocation of Static Type

- In addition to global declaration of variables or declaration of string-constants, you may explicitly ask for declaration of static variables.
- Explicit declaration of static variable also happens on the static area, and happens before main function starts.
- Example:

```
void fun(void);
int main(void) {
     for(int i = 0; i < 3; i++) {
       fun();
     }
     return 0;
}

void fun(void) {
     static int s = 30;
     int m = 10;
     s++;
     m++;
     printf("%d   %d\n", s, m);
     // Point one
}
```

- What is the program's output?
- Draw a memory diagram when program reaches point one for the second time.

Some Operations on Arrays

- Important operations on arrays:
  - Sort
  - Append
  - Insert,
  - Delete
  - Reverse
  - Search
  - Extract
  - Return subset
  - etc.
- To increase the efficiency of operation on arrays, we should minimize the number of the times that program needs to visit the elements of the array.

Data Scope and Lifetime

Data Scope and Lifetiem

- ***Scope*** is the extent to which your variables and functions are "*known*" and can be used throughout your program
- A variable which is declared **outside** the body of the program functions is a ***global declaration***
  - This type of variable are called global variables and their scope is from the declaration to the end of the ***file***
    - This scope is called ***global scope*** or ***file scope***
    - The lifetime of this variables is also from the point of declaration to the end of the program

Data Scope and Lifetime (cont'd)

- A variables which are declared ***inside*** of a function or any block confined between opening and closing braces, { …}, are called ***local declaration***
  - The scope of the this type of declaration is from the declaration to the end of the ***block*** (i.e., until the closing brace)
    - This is called ***block scope*** or ***local scope***
  - The lifetime of this type of declaration is from point declaration until the function or block ends.

Data Scope and Lifetime

- It is possible to have a local object with the same name as a global object
  - In this case the local name takes precedence, and we say that the ***local declaration hides the global declaration***

- Syntax rule for multiple objects with the same name:
  - An identifier (i.e., variable name) matches the declaration that
    1. Precedes it
    2. Is closest to its use
    3. Is within its scope

- If a function prototype is not provided, its scope starts from the first line of its definition

A Simple Example:

```
#include <stdio.h>
int a = 90;          // global declaration of a

int main (void)
{
    int a = 34;     // local declaration of a
    int b;          // local declaration of b
    b = a + 2;
    int c = 68;
    for(int i =0; i < 5; i++)   // local i
    {
            int d = 50;             // local d

            // MORE CODE
             …
    }
    return 0;
 }
```

Data Scope and Lifetime: Another Example

```
1  const int a = 3;
2
3  void f(int a, int z);
4
5  void main()
6  {
7     int z = 7;
8     printf("a = %d z = %d\n", a, z);
9     {
10       int a = 9;
11       printf("a = %d z = %d\n", a, z);
12       f(z,a);
13       printf("a = %d z = %d\n", a, z);
14    }
15    f(z,a);
16    printf("a = %d z = %d\n", a, z);
17 }
18
19 void f(int a, int z)
20 {
21    int i = 1;
22    a = i + z;
23    {
24       int a = 2;
25       z *= a;
26    }
27 }
```

Scope of a

Scope of f()

Scope of z

Scope of a

Scope of a & z

Scope of i

Scope of a