Arrays and Strings in Assembly

CSE 2312
Computer Organization and Assembly Language Programming
Vassilis Athitsos
University of Texas at Arlington

Arrays

- Which of the following are true?
  A. An array is a memory address.
  B. An array is a pointer.

Arrays

- Which of the following are true?
  A. An array is a memory address.
  B. An array is a pointer.

- Both are true!
- Both are **partial** descriptions of what an array is.
- An array is a memory address marking the **beginning** of a piece of memory containing items of a specific type.
- Note: there is **no difference** between a memory address and a pointer, they are synonyms.

Arrays

- In C, you can declare an array explicitly, for example:

```
int a[10];
char * b = malloc(20);
```

Arrays

- In C, the compiler helps the programmer (to some extent) to use arrays the right way.

```
int num = 10;
int c = num[5];          Error, num is not an array.

char * my_string = malloc(20);
my_string(3, 2);         Error, my_string is not a function.
```

Arrays

- Even in C the compiler will not catch some errors.

```
int my_array = malloc(20 * sizeof(int));

int c = my_array[100];
```

Index goes beyond the length of the array, the compiler does not catch that.

```
free(my_array);
int d = my_array[2];
```

Accessing the array after memory has been deallocated, the compiler does not catch that.

Arrays

- In assembly, there are no variables and types.
- It is useful to use arrays and think of them as arrays.
- However, there is no explicit way to define arrays.
- It is the programmer's responsibility to make sure that what they think of as an array:
  - Is indeed an array.
  - Is used correctly.

Creating an Array

- Assembler directives can be used to create an array.
- Example 1: create an array of 3 integers.

```
my_array:
        .word 3298
        .word 1234567
        .word -9878
```

- The compiler makes sure that:
  - This array is stored **somewhere in memory** (the compiler chooses where, not us).
  - References to my_array will be replaced by references to the memory address where the array is stored.

MEMORY	
Address	Contents
…	…
…	…
???	???
???	???
???	???
4765468	???
???	???
???	???

my_array

Creating an Array

- Assembler directives can be used to create an array.
- Example 1: create an array of 3 integers.

```
my_array:
        .word 3298
        .word 1234567
        .word -9878
```

my_array

- The compiler makes sure that:
  - This array is stored **somewhere in memory** (the compiler chooses where, not us).
  - References to my_array will be replaced by references to the memory address where the array is stored.

MEMORY	
Address	Contents
…	…
…	…
…	…
4765476	-9878
4765472	1234567
4765468	3298
…	…
…	…

Using the Array

```
        ldr r9, =my_array
        ldr r0, [r9, #8]
        bl print10
...
my_array:
        .word 3298
        .word 1234567
        .word -9878
```

- What does this do?

	MEMORY	
	Address	Contents
	…	…
	…	…
	…	…
	4765476	-9878
	4765472	1234567
my_array	4765468	3298
	…	…
	…	…

Using the Array

```
        ldr r9, =my_array
        ldr r0, [r9, #8]
        bl print10
...
my_array:
        .word 3298
        .word 1234567
        .word -9878
```

- What does this do?
- Prints my_array[2].

	MEMORY	
	Address	Contents
	…	…
	…	…
	…	…
	4765476	-9878
	4765472	1234567
my_array	4765468	3298
	…	…
	…	…

Second Example

```
sub sp, sp, #12
ldr r6, =3298
str r6, [sp, #0]
ldr r6, =1234567
str r6, [sp, #4]
ldr r6, =-9878
str r6, [sp, #8]
mov r8, sp
```

- At this point, register r8 contains the address of the array.
- Note: when the current function returns, and the stack pointer moves on top of r8, **array contents may be written over** by other functions.
- Until the current function returns, the array pointed to by r8 will be valid.

MEMORY	
Address	Contents
…	…
…	…
…	…
4765476	-9878
4765472	1234567
4765468 **r8**	3298
…	…
…	…

Compare to C

```
int foo(int a, int b)
{
  …
  int a[10];
  …
}
```

- Note: when the current function returns, array a[] does not exist any more.

MEMORY	
Address	Contents
…	…
…	…
…	…
4765476	-9878
4765472	1234567
4765468	3298
…	…
…	…

Using the Array

```
sub sp, sp, #12
ldr r6, =3298
str r6, [sp, #0]
ldr r6, =1234567
str r6, [sp, #4]
ldr r6, =-9878
str r6, [sp, #8]
mov r8, sp
...
```

- How do we print elements at position 0, 1, 2?

	MEMORY	
	Address	Contents
	...	...
	...	...
	...	...
	4765476	-9878
	4765472	1234567
r8	4765468	3298
	...	...
	...	...

Using the Array

```
sub sp, sp, #12
ldr r6, =3298
str r6, [sp, #0]
ldr r6, =1234567
str r6, [sp, #4]
ldr r6, =-9878
str r6, [sp, #8]
mov r8, sp
...
ldr r0, [r8, #0]
bl print10
ldr r0, [r8, #4]
bl print10
ldr r0, [r8, #8]
bl print10
```

	MEMORY	
	Address	Contents
	...	...
	...	...
	...	...
	4765476	-9878
	4765472	1234567
r8	4765468	3298
	...	...
	...	...

Using the Array

```
sub sp, sp, #12
ldr r6, =3298
str r6, [sp, #0]
ldr r6, =1234567
str r6, [sp, #4]
ldr r6, =-9878
str r6, [sp, #8]
mov r8, sp
…
```

	MEMORY	
	Address	Contents
	…	…
	…	…
	…	…
	4765476	-9878
	4765472	1234567
r8	4765468	3298
	…	…
	…	…

- How do we set r9 to be the sum of all elements in the array?

Using the Array

```
sub sp, sp, #12
ldr r6, =3298
str r6, [sp, #0]
ldr r6, =1234567
str r6, [sp, #4]
ldr r6, =-9878
str r6, [sp, #8]
mov r8, sp
...
ldr r9, [r8, #0]
ldr r0, [r8, #4]
add r9, r9, r0
ldr r0, [r8, #8]
add r9, r9, r0
```

	MEMORY	
	Address	Contents
	...	...
	...	...
	...	...
	4765476	-9878
	4765472	1234567
r8	4765468	3298
	...	...
	...	...

Using the Array

```
sub sp, sp, #12
ldr r6, =3298
str r6, [sp, #0]
ldr r6, =1234567
str r6, [sp, #4]
ldr r6, =-9878
str r6, [sp, #8]
mov r8, sp
...
```

- How do we write a function array_sum that returns the sum of all elements in the array?

	MEMORY	
	Address	Contents
	...	...
	...	...
	...	...
	4765476	-9878
	4765472	1234567
r8	4765468	3298
	...	...
	...	...

Using the Array

```
sub sp, sp, #12
ldr r6, =3298
str r6, [sp, #0]
ldr r6, =1234567
str r6, [sp, #4]
ldr r6, =-9878
str r6, [sp, #8]
mov r8, sp
...
```

- How do we write a function array_sum that returns the sum of all elements in the array?
- What arguments does the function need?

	MEMORY	
	Address	Contents
	...	...
	...	...
	...	...
	4765476	-9878
	4765472	1234567
r8	4765468	3298
	...	...
	...	...

Using the Array

```
sub sp, sp, #12
ldr r6, =3298
str r6, [sp, #0]
ldr r6, =1234567
str r6, [sp, #4]
ldr r6, =-9878
str r6, [sp, #8]
mov r8, sp
...
```

	MEMORY	
	Address	Contents
	...	...
	...	...
	...	...
	4765476	-9878
	4765472	1234567
r8	4765468	3298
	...	...
	...	...

- How do we write a function array_sum that returns the sum of all elements in the array?
- What arguments does the function need?
- The array itself (i.e., the memory address).
- **The length of the array.** Very important, functions have no way of knowing the length of an array.

array_sum

```
array_sum:
        push {r4, r5, r6, r7, lr}
        mov r4, r0
        mov r0, #0
        mov r5, #0
array_sum_loop:
        cmp r5, r1
        bge array_sum_exit
        lsl r7, r5, #2
        ldr r6, [r4, r7]
        add r0, r0, r6
        add r5, r5, #1
        b array_sum_loop
array_sum_exit:
        pop {r4, r5, r6, r7, lr}
        bx lr
```

	MEMORY	
	Address	Contents
	…	…
	…	…
	…	…
	4765476	-9878
	4765472	1234567
r4	4765468	3298
	…	…
	…	…

array_sum

```
array_sum:
        push {r4, r5, r6, r7, lr}
        mov r4, r0
        mov r0, #0
        mov r5, #0
array_sum_loop:
        cmp r5, r1
        bge array_sum_exit
        lsl r7, r5, #2
        ldr r6, [r4, r7]
        add r0, r0, r6
        add r5, r5, #1
        b array_sum_loop
array_sum_exit:
        pop {r4, r5, r6, r7, lr}
        bx lr
```

Why do we do this?

	MEMORY	
	Address	Contents
	…	…
	…	…
	…	…
	4765476	-9878
	4765472	1234567
r4	4765468	3298
	…	…
	…	…

array_sum

```
array_sum:
        push {r4, r5, r6, r7, lr}
        mov r4, r0
        mov r0, #0
        mov r5, #0
array_sum_loop:
        cmp r5, r1
        bge array_sum_exit
        lsl r7, r5, #2
        ldr r6, [r4, r7]
        add r0, r0, r6
        add r5, r5, #1
        b array_sum_loop
array_sum_exit:
        pop {r4, r5, r6, r7, lr}
        bx lr
```

Why do we do this?
r0 contains first argument, but will also contain the result.
We copy the argument to r4, so that we can put the result on r0.

MEMORY	
Address	Contents
…	…
…	…
…	…
4765476	-9878
4765472	1234567
r4 4765468	3298
…	…
…	…

Using array_sum

```
sub sp, sp, #12
ldr r6, =3298
str r6, [sp, #0]
ldr r6, =1234567
str r6, [sp, #4]
ldr r6, =-9878
str r6, [sp, #8]
mov r8, sp
…
```

- How do we call array_sum from here?

	MEMORY	
	Address	Contents
	…	…
	…	…
	…	…
	4765476	-9878
	4765472	1234567
r8	4765468	3298
	…	…
	…	…

Using array_sum

```
sub sp, sp, #12
ldr r6, =3298
str r6, [sp, #0]
ldr r6, =1234567
str r6, [sp, #4]
ldr r6, =-9878
str r6, [sp, #8]
mov r8, sp
…
```

- How do we call array_sum from here?

```
mov r0, r8
mov r1, #3
bl array_sum
```

	MEMORY	
	Address	Contents
	…	…
	…	…
	…	…
	4765476	-9878
	4765472	1234567
r8	4765468	3298
	…	…
	…	…

array_sum

```
array_sum:
        push {r4, r5, r6, r7, lr}
        mov r4, r0
        mov r0, #0
        mov r5, #0
array_sum_loop:
        cmp r5, r1
        bge array_sum_exit
        lsl r7, r5, #2
        ldr r6, [r4, r7]
        add r0, r0, r6
        add r5, r5, #1
        b array_sum_loop
array_sum_exit:
        pop {r4, r5, r6, r7, lr}
        bx lr
```

Note:

- Function array_sum computes the sum of an array of 32-bit integers.
- It has no way of knowing/ensuring that the input array is indeed an array of 32-bit integers.
- It has no way of knowing that the length (passed as an argument in r1) is correct.
- It is the responsibility of the programmer to avoid mistakes.

MEMORY		
	Address	Contents
	…	…
	…	…
	…	…
	4765476	-9878
	4765472	1234567
r4	4765468	3298
	…	…
	…	…

Possible Errors

```
my_array:
        .word 3298
        .word 1234567
        .word -9878
…
        bl my_array
```

- You are asking the program to execute function my_array, but my_array is a string, not a function.
- C would not allow that, assembly does allow it.
- Instruction bl wants a memory address, doesn't care what you give it.
- Needless to say, this is usually NOT something you would do on purpose, it is a bug.

MEMORY	
Address	Contents
…	…
…	…
…	…
4765476	-9878
4765472	1234567
4765468	3298
…	…
…	…

my_array

Possible Errors

```
my_array:
        .word 3298
        .word 1234567
        .word -9878
…
        ldr r6, =my_array
        ldr r7, [r6, #2]
```

- What is wrong with this code?

MEMORY	
Address	Contents
…	…
…	…
…	…
4765476	-9878
4765472	1234567
my_array 4765468	3298
…	…
…	…

Possible Errors

```
my_array:
        .word 3298
        .word 1234567
        .word -9878
…
        ldr r6, =my_array
        ldr r7, [r6, #2]
```

- What is wrong with this code?
- Presumably we want to put on r7 the element at position 2 of the array.
- We need to use this:

```
        ldr r7, [r6, #8]
```

MEMORY	
Address	Contents
…	…
…	…
…	…
4765476	-9878
4765472	1234567
my_array 4765468	3298
…	…
…	…

Strings

- Which of the following are true?
  A. A string is a memory address.
  B. A string is a pointer.
  C. A string is an array of characters.

Strings

- Which of the following are true?
  A. A string is a memory address.
  B. A string is a pointer.
  C. A string is an array of characters.

- All three are true.
- All of them are **partial** descriptions of what a string is.
- Full description: a string is an array of characters (i.e., an array of 8-bit ASCII codes), that contains ASCII code 0 as its last character.
- This definition is **the same** in both C and assembly.

Creating a String

- Assembler directives can be used to create a string.
- Example 1:

```
string1:
        .asciz "Hello"
```

- The compiler makes sure that:
  - This string is stored **somewhere in memory** (the compiler chooses where, not us).
  - References to string1 will be replaced by references to the memory address where the array is stored.

MEMORY	
Address	Contents
…	…
…	…
???	???
???	???
???	???
???	???
???	???
string1 4765468	???
???	???

Creating a String

- Assembler directives can be used to create a string.
- Example 1:

```
string1:
        .asciz "Hello"
```

- The compiler makes sure that:
  - This string is stored **somewhere in memory** (the compiler chooses where, not us).
  - References to string1 will be replaced by references to the memory address where the array is stored.

MEMORY	
Address	Contents
…	…
…	…
4765473	'\0'
4765472	'o'
4765471	'l'
4765470	'l'
4765469	'e'
4765468	'H'
…	…

string1

Creating a String

- Assembler directives can be used to create a string.
- Example 1:

```
string1:
        .ascii "105-c"
        .byte 0x00
```

MEMORY	
Address	Contents
…	…
…	…
4765473	???
4765472	???
4765471	???
4765470	???
4765469	???
string1 4765468	???
…	???

Creating a String

- Assembler directives can be used to create a string.
- Example 1:

```
string1:
        .ascii "105-c"
        .byte 0x00
```

- Common question:
  - Since the last character of the string is 0, how can we put an actual zero in the string?

MEMORY	
Address	Contents
…	…
…	…
4765473	'\0'
4765472	'c'
4765471	'-'
4765470	'5'
4765469	'0'
string1 4765468	'1'
…	…

Creating a String

- Assembler directives can be used to create a string.
- Example 1:

```
string1:
        .ascii "105-c"
        .byte 0x00
```

- Common question:
  - Since the last character of the string is 0, how can we put an actual zero in the string?
- Answer: character '0' is **not** character 0.
  - Character '0' has ASCII code 48.
  - Character 0 (also written '\0') has ASCII code 0.

MEMORY	
Address	Contents
…	…
…	…
4765473	'\0'
4765472	'c'
4765471	'-'
4765470	'5'
4765469	'0'
4765468	'1'
…	…

string1

Using a String

```
        lrd r4, =0x101f1000
        ldr r6, string1
        ldr r7, [r6, #3]
        str r7, [r4]

string1:
        .ascii "105-c"
        .byte 0x00
```

- What does this code do?

MEMORY	
Address	Contents
…	…
…	…
4765473	'\0'
4765472	'c'
4765471	'-'
4765470	'5'
4765469	'0'
string1 4765468	'1'
…	…

Using a String

```
        lrd r4, =0x101f1000
        ldr r6, string1
        ldr r7, [r6, #3]
        str r7, [r4]

string1:
        .ascii "105-c"
        .byte 0x00
```

- What does this code do?
  - It prints '-'.

MEMORY	
Address	Contents
…	…
…	…
4765473	'\0'
4765472	'c'
4765471	'-'
4765470	'5'
4765469	'0'
string1 4765468	'1'
…	…

Length of a String

```
        lrd r4, =0x101f1000
        ldr r6, string1
string1:
        .ascii "105-c"
        .byte 0x00
```

- How can we write a function that computes the length of a string?

	MEMORY	
	Address	Contents
	…	…
	…	…
	4765473	'\0'
	4765472	'c'
	4765471	'-'
	4765470	'5'
	4765469	'0'
string1	4765468	'1'
	…	…

Length of a String

```
        lrd r4, =0x101f1000
        ldr r6, string1

string1:
        .ascii "105-c"
        .byte 0x00
```

- How can we write a function that computes the length of a string?
- What arguments does it need?

	MEMORY	
	Address	Contents
	…	…
	…	…
	4765473	'\0'
	4765472	'c'
	4765471	'-'
	4765470	'5'
	4765469	'0'
string1	4765468	'1'
	…	…

Length of a String

```
        lrd r4, =0x101f1000
        ldr r6, string1

string1:
        .ascii "105-c"
        .byte 0x00
```

- How can we write a function that computes the length of a string?
- What arguments does it need?
  - Just the string itself (the memory address).

MEMORY	
Address	Contents
…	…
…	…
4765473	'\0'
4765472	'c'
4765471	'-'
4765470	'5'
4765469	'0'
string1 4765468	'1'
…	…

Length of a String

```
strlen:
        push {r4, r5, lr}

        mov r4, r0
        mov r0, #0

strlen_loop:
        ldrb r5, [r4, r0]
        cmp r5, #0
        beq strlen_exit
        add r0, r0, #1
        b strlen_loop

strlen_exit:
        pop {r4, r5, lr}
        bx lr
```

MEMORY	
Address	Contents
…	…
…	…
4765473	'\0'
4765472	'c'
4765471	'-'
4765470	'5'
4765469	'0'
r4 4765468	'1'
…	…

Using strlen

```
        ldr r6, string1
        mov r0, r6
        bl strlen

string1:
        .ascii "105-c"
        .byte 0x00
```

MEMORY	
Address	Contents
…	…
…	…
4765473	'\0'
4765472	'c'
4765471	'-'
4765470	'5'
4765469	'0'
string1 4765468	'1'
…	…

Find a Character

```
        lrd r4, =0x101f1000
        ldr r6, string1
        ldr r7, [r6, #3]
        str r7, [4]

string1:
        .ascii "105-c"
        .byte 0x00
```

- How can we write a function that finds the first occurrence of a character?
- Arguments?

MEMORY	
Address	Contents
…	…
…	…
4765473	'\0'
4765472	'c'
4765471	'-'
4765470	'5'
4765469	'0'
string1 4765468	'1'
…	…

Find a Character

```
strfind:
        push {r4, r5, lr}
        mov r4, r0
        mov r0, #0
strfind_loop:
        ldrb r5, [r4, r0]
        cmp r5, #0
        moveq r0, #-1
        beq strfind_exit
        cmp r5, r1
        beq strfind_exit
        add r0, r0, #1
        b strfind_loop
strfind_exit:
        pop {r4, r5, lr}
        bx lr
```

MEMORY	
Address	Contents
…	…
…	…
4765473	'\0'
4765472	'c'
4765471	'-'
4765470	'5'
4765469	'0'
r4 4765468	'1'
…	…