What you will learn.

In this chapter you will learn:
1) A third type of memory, heap memory, and how to allocate and use it.
2) The definition of an array and how to implement and access elements using assembly.
3) How to allocate an array in stack memory, on the program stack, or in heap memory, and why arrays are most commonly allocated on heap memory.
4) How to use array addresses to access and print elements in an array.
5) The Bubble Sort, and how to implement this sort in assembly.

Chapter 9 Arrays

In a HLL, an array is a multi-valued variable: a single array variable can contain many values. Arrays in MIPS assembly will be similar; however, the abstraction of an array is very much constrained in assembly. In MIPS assembly an array is implemented by storing multiple values in contiguous areas of memory and accessing each value in the array as an offset of the array value. This chapter will cover how to implement and use arrays in MIPS assembly.

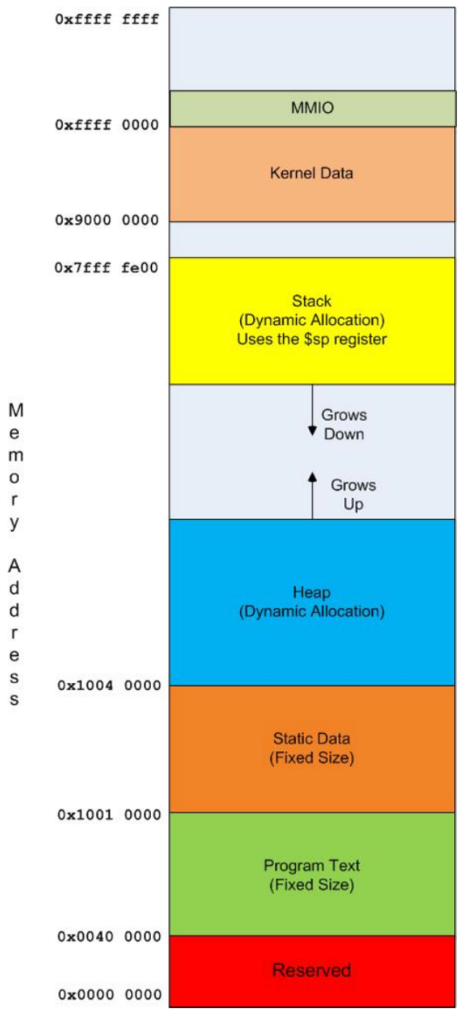
Chapter 9. 1 Heap dynamic memory

Before beginning a discussion of arrays, a third type of program data memory is introduced. So far in this text static (data) and stack dynamic memory have been discussed. The third type of memory introduced here is heap dynamic memory.

Arrays can exist in any type of memory, static, stack, or heap. However, when they are stored in static or stack memory, their size is fixed when the program is compiled and cannot change. Most programs that use arrays will need arrays that adjust in size depending on the size of the input data, and this requires heap memory.

Chapter 9.1. 1 What is heap memory

Heap memory is shown to the right (blue). As this figure shows, the heap memory segment begins in the program process space immediately after static memory and grows upward in memory until theoretically it reaches the stack memory segment. Most systems limit the amount of heap a process can request to protect against incorrect programs that might be allocating heap inappropriately. Generally, these limits on heap size can be increased if they need to be larger than the default (for example, the Java interpreter can be run with the -Xms or -Xmx parameters to change the heap size).



Heap memory is dynamic, like stack memory, in that it is allocated at run time. However, unlike stack memory which is automatically allocated and de-allocated when subprograms are entered and exited, heap memory is allocated based on a request from the programmer, normally using a *new* operator. The new operator allocates the requested amount of memory and initializes it to a default value (normally zero). Because heap memory is allocated when requested by the user at run time, the amount of memory can be determined at run time.

Memory in the heap is generally de-allocated when it is no longer needed by the programmer. How this de-allocation is done is dependent on the environment in which the program is being run. For example in C/C++ memory is de-allocated using either the free() function call or the delete operator. Java, C#, and other modern languages use memory managers which automatically free the memory when it is no longer used.

The biggest issue with heap memory is since it is allocated and de-allocated in non-fixed sized pieces, as the memory is de-allocated it leaves numerous holes in the memory that are of various sizes. This makes memory allocation and de-allocation difficult to deal with. Heap memory is much more complicated to manage than static and stack memory, and entire books have been written on heap management.

Because of the complexity of dealing with managing the de-allocation of heap memory, this text will only deal with the allocation of the memory. Any heap memory that is allocated cannot be reused.

Chapter 9.1. 2 Allocating heap memory example – PromptString

The first example subprogram presented here for using heap memory is a function that allows a programmer to prompt for strings without having to create a blank string variable in the `.data` section of the program. The subprogram first allocates a string variable large enough to hold the string the user is being prompted for, and then uses the syscall service 8 to read a value into that string.

```
.text
main:

    la $a0, prompt1     # Read and print first string
    li $a1, 80
    jal PromptString
    move $a0, $v0
    jal PrintString

    la $a0, prompt2     # Read and print second string
    li $a1, 80
    jal PromptString
    move $a0, $v0
    jal PrintString

    jal Exit

.data
    prompt1: .asciiz "Enter the first string: "
    prompt2: .asciiz "Enter the second string: "
```

```
# Subprogram:     PromptString
# Author:   Charles Kann
# Purpose:  To prompt for a string, allocate the string
#           and return the string to the calling subprogram.
# Input:    $a0 - The prompt
#           $a1 - The maximum size of the string
# Output:   $v0 - The address of the user entered string

.text
PromptString:
    addi $sp, $sp, -12  # Push stack
    sw $ra, 0($sp)
    sw $a1, 4($sp)
    sw $s0, 8($sp)

    li $v0, 4           # Print the prompt
    syscall             # in the function, so we know $a0 still has
                        # the pointer to the prompt.

    li $v0, 9           # Allocate memory
    lw $a0, 4($sp)
    syscall
    move $s0, $v0

    move $a0, $s0       # Read the string
    li $v0, 8
    lw $a1, 4($sp)
    syscall

    move $v0, $s0       # Save string address to return

    lw $ra, 0($sp)      # Pop stack
    lw $s0, 8($sp)
    addi $sp, $sp, 12
    jr $ra

.include "utils.asm"
```

Program 0-1: PromptString subprogram showing heap allocation

Chapter 9.1. 3 Commentary on PromptString Subprogram

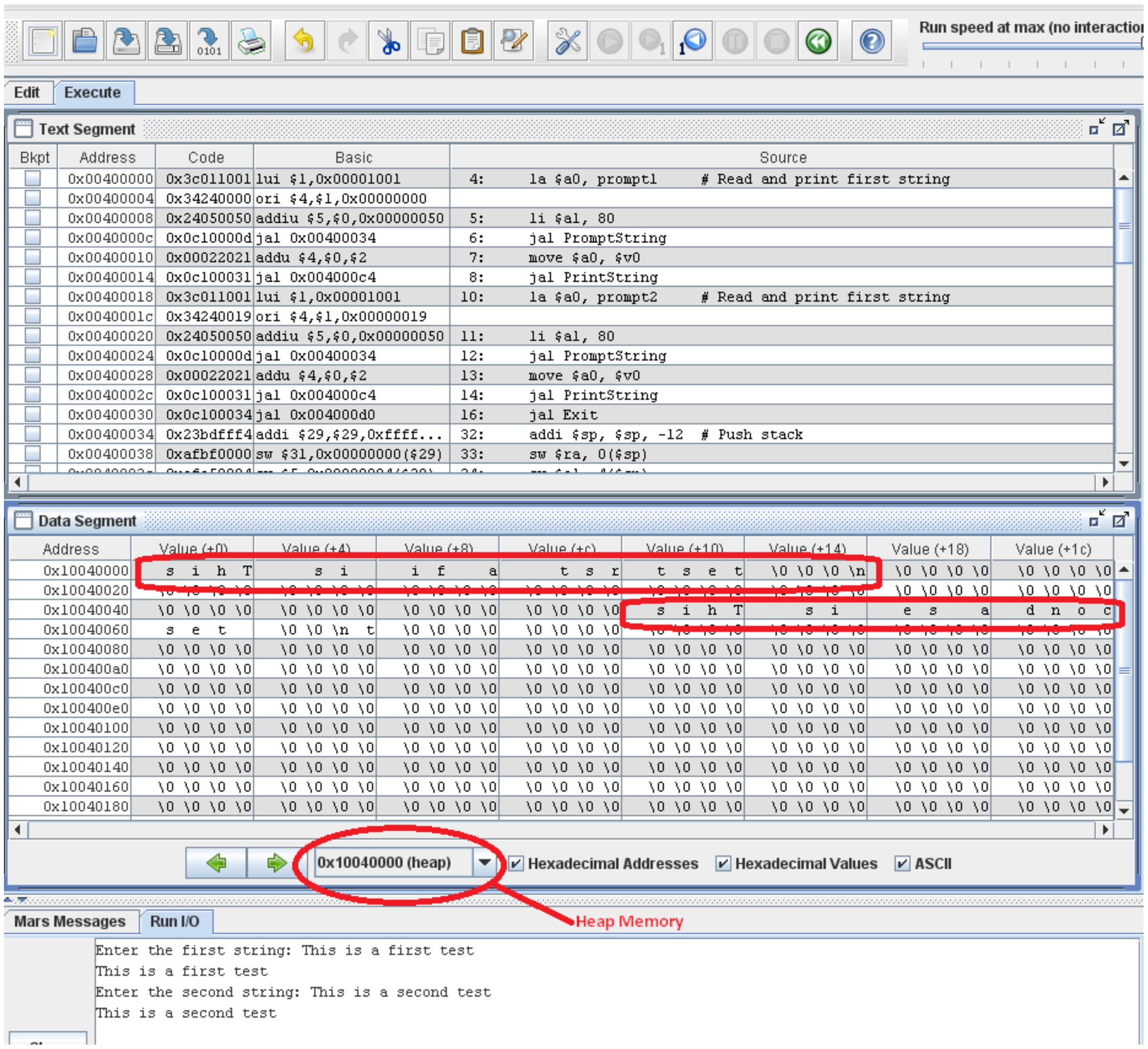
1) To allocate heap memory, the syscall service 9 is used. The address of the memory returned from this heap allocation syscall is in `$v0`. `$v0` is moved to `$a0` to be used in the syscall service 8 to read a string. The address of the memory containing the string is now in `$a0` and is moved to `$v0` to be returned to the main subprogram.

2) Data which is expected to be unchanged across subprogram calls (including syscall) should always be stored in a save register (`$s0` in this example), or on the stack (`$a1` in this example). Do not use any other registers (such as temporary registers like `$t0`) or memory as the values cannot be guaranteed across subprogram calls.

3) The value of `$s0` is saved when this subprogram is entered and restored to its original value when the subprogram is exited. All save registers have this behavior.

4) The main subprogram in this example will show two strings being read. This is to show how the allocated strings exist in heap memory. In the MARS screen shot below, note that the heap memory is now being displayed. In the heap memory the two strings entered ("This is a first test", and "This is a second test") are shown, with each taking up 80 bytes of memory.

5)



Bkpt	Address	Code	Basic	Source
☐	0x00400000	0x3c011001	lui $1,0x00001001	4: la $a0, prompt1 # Read and print first string
☐	0x00400004	0x34240000	ori $4,$1,0x00000000	
☐	0x00400008	0x24050050	addiu $5,$0,0x00000050	5: li $a1, 80
☐	0x0040000c	0x0c10000d	jal 0x00400034	6: jal PromptString
☐	0x00400010	0x00022021	addu $4,$0,$2	7: move $a0, $v0
☐	0x00400014	0x0c100031	jal 0x004000c4	8: jal PrintString
☐	0x00400018	0x3c011001	lui $1,0x00001001	10: la $a0, prompt2 # Read and print first string
☐	0x0040001c	0x34240019	ori $4,$1,0x00000019	
☐	0x00400020	0x24050050	addiu $5,$0,0x00000050	11: li $a1, 80
☐	0x00400024	0x0c10000d	jal 0x00400034	12: jal PromptString
☐	0x00400028	0x00022021	addu $4,$0,$2	13: move $a0, $v0
☐	0x0040002c	0x0c100031	jal 0x004000c4	14: jal PrintString
☐	0x00400030	0x0c100034	jal 0x004000d0	16: jal Exit
☐	0x00400034	0x23bdfff4	addi $29,$29,0xffff...	32: addi $sp, $sp, -12 # Push stack
☐	0x00400038	0xafbf0000	sw $31,0x00000000($29)	33: sw $ra, 0($sp)

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10040000	s i h T	s i	i f a	t s r	t s e t	\0 \0 \0 \n	\0 \0 \0 \0	\0 \0 \0 \0
0x10040020	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10040040	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	s i h T	s i	e s a	d n o c
0x10040060	s e t	\0 \0 \n t	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10040080	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x100400a0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x100400c0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x100400e0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10040100	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10040120	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10040140	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10040160	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10040180	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0

0x10040000 (heap) [✓] Hexadecimal Addresses [✓] Hexadecimal Values [✓] ASCII

```
Enter the first string: This is a first test
This is a first test
Enter the second string: This is a second test
This is a second test
```

Figure 0-1: Heap memory example

Chapter 9. 2 Array Definition and creation in Assembly

Most readers of this text will be familiar with the concept of arrays as they are used in a HLL. So, this chapter will not cover their use, but how arrays are implemented and elements in the array accessed in assembly. Most HLL go to great pains to hide these details from the programmer, with good reason. When programmers deal with the details, they often make mistakes that have serious consequences to the correctness of their programs: mistakes that lead to serious correctness problems with their programs, and bugs that can often make it very difficult to locate and fix.

But even though the details of arrays are hidden in most HLL, the details affect how HLL implement array abstractions, and the proper understanding of arrays can help prevent programmers from developing inappropriate metaphors that lead to program issues. Misusing object slicing in C++ or allocating and attempting to use arrays of null objects in Java are issues that can arise if a programmer does not understand true nature of an array.

The following definition of an array will be used in this chapter. **An array is a multivalued variable stored in a contiguous area of memory that contains elements that are all the same size**. Some programmers will find that this definition does not fit the definition of arrays in the HLL language which they use. This is a result of the HLL adding layers of abstraction, such as Perl associative array (which are really hash tables) or Java object arrays or ArrayList. These HLL arrays are always hiding some abstraction and knowing what an array actually is, can help with the understanding of how the HLL is manipulating the array.

The definition of an array becomes apparent when the mechanics of accessing elements in an array is explained. The minimum data needed to define an array consists of a variable which contains the address of the start of the array, the size of each element, and the space to store the elements. For example, an array based at address 0x10010044 and containing 5 32-bit integers is shown in Figure 9-2.

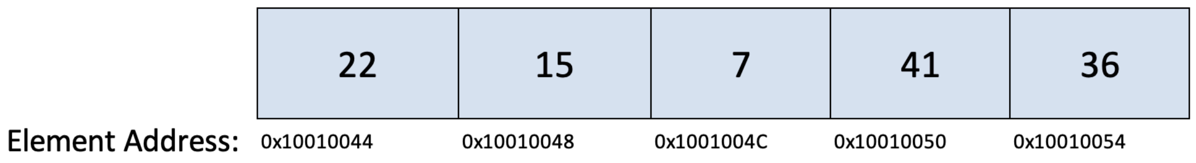


Figure 0-2: Array implementation

To access any element in the array, the element address is calculated by the following formula, and the element valued is loaded from that address.

```
elemAddress = basePtr + index * size
```

where:

- `elemAddress` is the address of (or pointer to) the element to be used
- `basePtr` is the address of the array variable
- `index` is the index for the element (using 0 based arrays)
- `size` is the size of each element

So, to load the element at index 0, the `elemAddress` is just (0x10010044 + (0 * 4)) = 0x10010044, or the `basePtr` for the array[1]. Likewise, to load an element at index 2, the `elemAddress` is (0x10010044 + (2 * 4)) = 0x1001004C.

[1] This calculation of the array address will make it apparent to many readers why arrays in many languages are zero based (the first element is 0), rather than the more intuitive concept of arrays being 1 based (the first element is 1). When thought of in terms of array addressing, the first element in the array is at the base address for the array (`basePtr + 0`), and so the number of the elements has more to do with how arrays are implemented, than in semantic considerations of what the elements numbers mean.

Two array examples follow. The first creates an array named grades, which will store 10 elements, each 4 bytes, aligned on word boundaries. The second creates an array named id of 10 bytes. Note that no alignment is specified, so the bytes can cross word boundaries.

```
.data
.align 2
grades: .space 40
id: .space 10
```

To access a grade element in the array grades, grade 0 would be at the `basePtr`, grade 1 would be at `basePtr`+4, grade 2 would be at `basePtr` + 8, etc. The following code fragment shows how grade 2 could be accessed in MIPS assembly code:

```
    addi $t0, 2             # set element number 2
    sll $t0, $t0, 2         # multiply $t0 by 4 (size) to get the offset
    la $t1, basePtr         # $t1 is the base of the array
    add $t0, $t0, $t1       # basePtr + (index * size)
    lw $t2, 0($t0)          # load element 2 into $t2
```

Addressing of arrays is not complicated, but it does require that the programmer keep in mind what an address is verses a value, and to know calculate an array offset.

Chapter 9.2. 1 Allocating arrays in memory

In some languages, such as Java, arrays can only be allocated on the heap. Others, such as C/C++ or C#, allow arrays of some types to be allocated anywhere in memory. In MIPS assembly, arrays can be allocated in any part of memory. However, remember that arrays allocated in the static data region or using a stack must be fixed size, with the size fixed at assembly time. Only heap allocated arrays can have their size set at run time.

To allocate an array in **static data**, a label is defined to give the base address of the array, and enough space for the array elements is allocated. Note also that the array must consider any alignment consideration (e.g., words must fall on word boundaries). The following code fragment allocates an array of 10 integer words in the data segment.

```
    .data
    .align 2
    array: .space 40
```

To allocate an array on the **stack**, the `$sp` is adjusted to allow space on the stack for the array. In the case of the stack, there is no equivalent to the `.align 2` assembler directive used above for static data, so the programmer is responsible for making sure any stack memory is properly aligned. The following code fragment allocates an array of 10 integer words on the stack after the `$ra` register.

```
    addi $sp, $sp, -44
    sw $ra, 0(sp)
    # array begins at 4($sp)
```

Finally, to allocate an array on the **heap**, the number of items to allocate is multiplied by the size of each element to obtain the amount of memory to allocate. A subprogram to do this, called AllocateArray, is shown below.

```
# Subprogram:     AllocateArray
# Purpose:        To allocate an array of $a0 items, each of size $a1.

# Author:   Charles Kann
# Input:    $a0 - the number of items in the array
#           $a1 - the size of each item
# Output:   $v0 - Address of the array allocated

      AllocateArray:
          addi $sp, $sp, -4
          sw $ra, 0($sp)

          mul $a0, $a0, $a1
          li $v0, 9
          syscall

          lw $ra, 0($sp)
          addi $sp, $sp, 4
          jr $ra
```

Program 0-2: AllocateArray subprogram

Chapter 9. 3 Printing an Array

This first program presented here shows how to access arrays by creating a PrintIntArray subprogram that prints the elements in an integer array. Two variables are passed into the subprogram, `$a0`, which is the base address of the array, and `$a1`, which is the number of elements to print. The subprogram processes the array in a counter loop and prints out each element followed by a ",". The pseudocode for this subprogram follows.

```
      Subprogram PrintIntArray(array, size)
      {
          print("[")
          for (int i = 0; i < size; i++)
          {
              print("," + array[i])
          }
          print("]")
      }
```

The following is the subprogram in assembly, along with a test main program to show how to use it.

```
      .text
      .globl main
      main:
          la $a0, array_base
          lw $a1, array_size
          jal PrintIntArray
          jal Exit
```

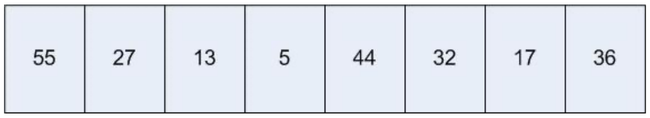
```
.data
    array_size: .word 5
    array_base:
            .word 12
            .word 7
            .word 3
            .word 5
            .word 11

# Subprogram: PrintIntArray
# Purpose: print an array of ints
# inputs: $a0 - the base address of the array
#         $a1 - the size of the array

.text
PrintIntArray:
    addi $sp, $sp, -16          # Stack record
    sw $ra, 0($sp)
    sw $s0, 4($sp)
    sw $s1, 8($sp)
    sw $s2, 12($sp)
    move $s0, $a0               # save the base of the array to $s0

    # initialization for counter loop
    # $s1 is the ending index of the loop
    # $s2 is the loop counter
    move $s1, $a1
    move $s2, $zero
    la $a0 open_bracket         # print open bracket
    jal PrintString

loop:
    # check ending condition
    sge $t0, $s2, $s1
    bnez $t0, end_loop

        sll $t0, $s2, 2         # Multiply the loop counter by 4 to get
                                # offset (each element is 4 big)
        add $t0, $t0, $s0       # address of next array element
        lw $a1, 0($t0)          # Next array element
        la $a0, comma
        jal PrintInt            # print the integer from array

        addi $s2, $s2, 1        #increment $s0
        b loop
end_loop:

    li $v0, 4                   # print close bracket
    la $a0, close_bracket
    syscall

    lw $ra, 0($sp)
    lw $s0, 4($sp)
    lw $s1, 8($sp)
    lw $s2, 12($sp)             # restore stack and return
    addi $sp, $sp, 16
    jr $ra
```

```
.data
    open_bracket:       .asciiz "["
    close_bracket:      .asciiz "]"
    comma:        .asciiz ","
.include "utils.asm"
```

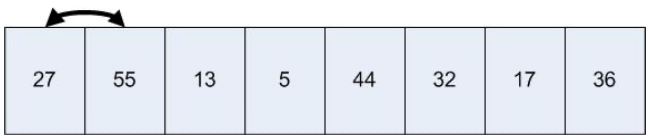
Program 0-3: Printing an array of integers

Chapter 9. 4 Bubble Sort

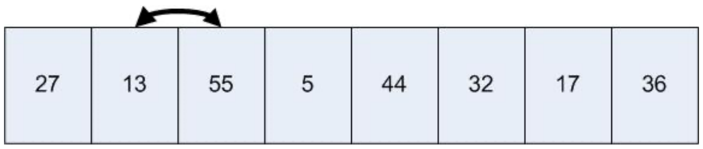
Sorting is the process of arranging data in an ascending or descending order. This example will introduce an algorithm, the Bubble Sort, for sorting integer data in an array. Consider for example the following array containing integer values.



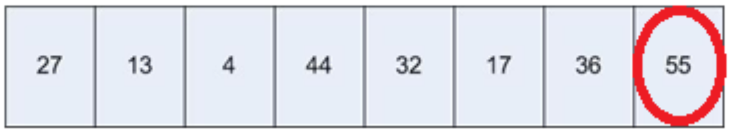
The sort is carried out in two loops. The inner loop passes once through the data comparing elements in the array and swapping them if they are not in the correct order. For example, element 0 (55) is compared to element 1 (27), and they are swapped since 55 > 27.



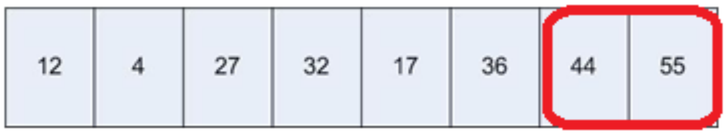
Next element 1 (now 55) is compared with element 2 (13), and they are swapped since 55 > 13.



This process continues until a complete pass has been made through the array. At the end of the inner loop, the largest value of the array is at the end of the array and in its correct position. The array would look as follows.



An outer loop now runs which repeats the inner loop, and the second largest value moves to the correct position, as shown below.



Repeating this outer loop for all elements results in the array being sorted in ascending order. Pseudocode for this algorithm follows.

```
for (int i = 0; i < size-1; i++)
{
    for (int j = 0; j < ((size-1)-i); j++)
    {
         if (data[j] > data[j+1])
         {
             swap(data, j, j+1)
         }
    }
}

swap(data, i, j)
    int tmp = data[i];
    data[i] = data[j];
    data[j] = tmp;
}
```

Chapter 9.3. 1 Bubble Sort in MIPS assembly

The following assembly program implements the Bubble Sort matching the pseudocode algorithm in the previous section.

```
.text
.globl main
main:
    la $a0, array_base
    lw $a1, array_size
    jal PrintIntArray

    la $a0, array_base
    lw $a1, array_size
    jal BubbleSort

    jal PrintNewLine
    la $a0, array_base
    lw $a1, array_size
    jal PrintIntArray

    jal Exit
.data
    array_size: .word 8
    array_base:
            .word 55
            .word 27
            .word 13
            .word 5
            .word 44
            .word 32
            .word 17
            .word 36
# Subproram:      Bubble Sort
# Purpose:        Sort data using a Bubble Sort algorithm
# Input Params:   $a0 - array
#           $a1 - array size
# Register conventions:
#           $s0 - array base
#           $s1 - array size
#           $s2 - outer loop counter
#           $s3 - inner loop counter
.text
BubbleSort:
    addi $sp, $sp, -20  # save stack information
    sw $ra, 0($sp)
    sw $s0, 4($sp)      # need to keep and restore save registers
    sw $s1, 8($sp)
    sw $s2, 12($sp)
    sw $s3, 16($sp)

    move $s0, $a0
    move $s1, $a1

    addi $s2, $zero, 0  #outer loop counter
    OuterLoop:
        addi $t1, $s1, -1
        slt $t0, $s2, $t1
        beqz $t0, EndOuterLoop

        addi $s3, $zero, 0 #inner loop counter
        InnerLoop:
            addi $t1, $s1, -1
            sub $t1, $t1, $s2
            slt $t0, $s3, $t1
            beqz $t0, EndInnerLoop

            sll $t4, $s3, 2   # load data[j].  Note offset is 4 bytes
            add $t5, $s0, $t4
            lw $t2, 0($t5)

            addi $t6, $t5, 4  # load data[j+1]
            lw $t3, 0($t6)

            sgt $t0, $t2, $t3
            beqz $t0, NotGreater
                move $a0, $s0
                move $a1, $s3
                addi $t0, $s3, 1
                move $a2, $t0
                jal Swap      # t5 is &data[j], t6 is &data[j+1]

            NotGreater:
            addi $s3, $s3, 1
            b InnerLoop
        EndInnerLoop:

        addi $s2, $s2, 1
        b OuterLoop
    EndOuterLoop:
```

```
    lw $ra, 0($sp)      #restore stack information
    lw $s0, 4($sp)
    lw $s1, 8($sp)
    lw $s2, 12($sp)
    lw $s3, 16($sp)
    addi $sp, $sp 20
    jr $ra

# Subprogram:           swap
# Purpose:        to swap values in an array of integers
# Input parameters:     $a0 - the array containing elements to swap
#                 $a1 - index of element 1
#                 $a2 - index of elelemnt 2
# Side Effects:         Array is changed to swap element 1 and 2
Swap:
    sll $t0, $a1, 2     # calcualate address of element 1
    add $t0, $a0, $t0
    sll $t1, $a2, 2     # calculate address of element 2
    add $t1, $a0, $t1

    lw $t2, 0($t0)      #swap elements
    lw $t3, 0($t1)
    sw $t2, 0($t1)
    sw $t3, 0($t0)

    jr $ra

# Subprogram: PrintIntArray
# Purpose: print an array of ints
# inputs: $a0 - the base address of the array
#         $a1 - the size of the array
#
PrintIntArray:
    addi $sp, $sp, -16        # Stack record
    sw $ra, 0($sp)
    sw $s0, 4($sp)
    sw $s1, 8($sp)
    sw $s2, 12($sp)

    move $s0, $a0             # save the base of the array to $s0

    # Initialization for counter loop
    # $s1 is the ending index of the loop
    # $s2 is the loop counter
    move $s1, $a1
    move $s2, $zero

   la $a0, open_bracket       # print open bracket
   jal PrintString

loop:
    # check ending condition
    sge $t0, $s2, $s1
    bnez $t0, end_loop

        sll $t0, $s2, 2       # Multiply the loop counter by
                              # by 4 to get offset (each element
                                # is 4 big).
        add $t0, $t0, $s0       # address of next array element
        lw $a1, 0($t0)          # Next array element
        la $a0, comma
        jal PrintInt            # print the integer from array

        addi $s2, $s2, 1        #increment $s0
        b loop
end_loop:

    li $v0, 4                   # print close bracket
    la $a0, close_bracket
    syscall


    lw $ra, 0($sp)
    lw $s0, 4($sp)
    lw $s1, 8($sp)
    lw $s2, 12($sp)             # restore stack and return
    addi $sp, $sp, 16
    jr $ra

.data
    open_bracket:       .asciiz "["
    close_bracket:      .asciiz "]"
    comma:        .asciiz ","
.include "utils.asm"
```

Program 0-4: Bubble Sort

Chapter 9. 5 Summary

In this chapter an array was defined as a multivalued variable stored in a contiguous area of memory that contains elements that are all the same size. The chapter then showed why each point in this definition is important, and how this definition can be used to implement an array and access array elements. The implementation and access to the array was shown in several programs, such as printing the array and sorting the array.

In terms of why this understanding of the true nature of an array is important, most HLL implement extensions to the basic array type, and a programmer who does not understand these extensions in the language is likely to have situations arise where bugs are encountered that are poorly understood. Even concepts as simple as Object arrays in Java are strange because the initial value of all elements is set to null. This is often confusing to new students until it is realized that in Java the size of an Object is unknown until it is allocated, so the only thing which can be used as the actual element in the Object array is the reference.

Chapter 9. 6 Exercises

1) Change the PrintIntArray subprogram so that it prints the array from the last element to the first element.

2) The pseudocode below converts an input value of a single decimal number from $1 \leq n \leq 15$ into a single hexadecimal digit. Translate this pseudocode into MIPS assembly.

```
String a[16]
a[0] = "0x0"
a[1] = "0x1"
a[2] = "0x2"
a[3] = "0x3"
a[4] = "0x4"
a[5] = "0x5"
a[6] = "0x6"
a[7] = "0x7"
a[8] = "0x8"
a[9] = "0x9"
a[10] = "0xa"
a[11] = "0xb"
a[12] = "0xc"
a[13] = "0xd"
a[14] = "0xe"
a[15] = "0xf"

int i = prompt("Enter a number from 0 to 15 ")
print("your number is " + a[i]
```