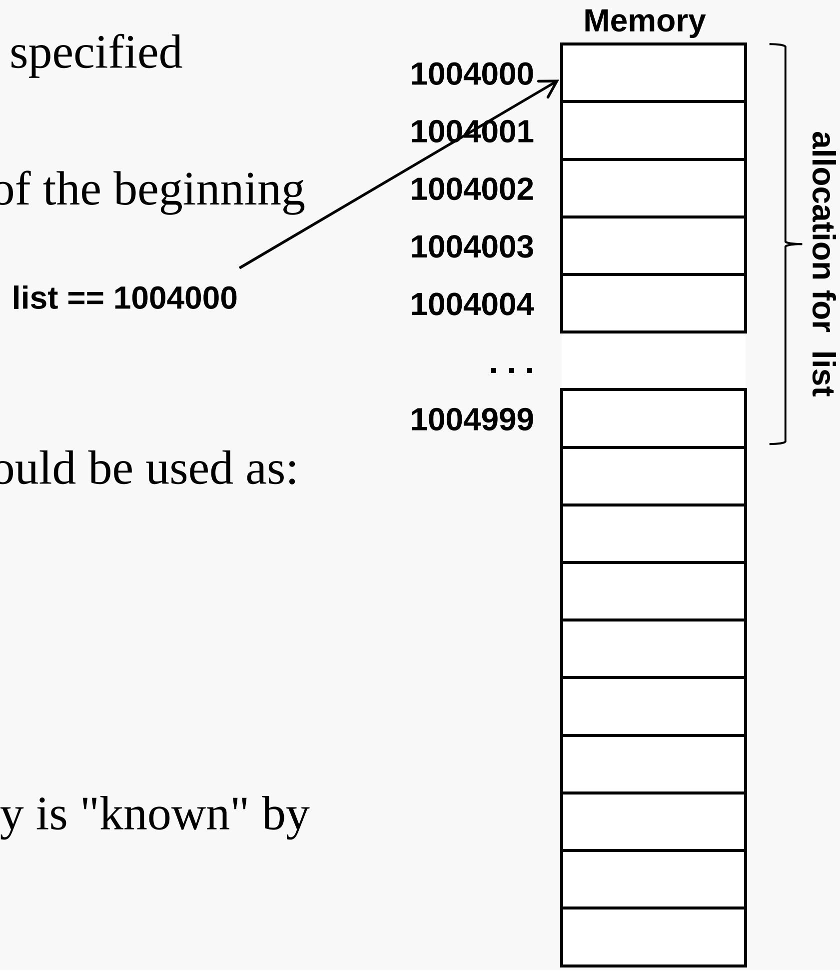
# Array Declaration and Storage Allocation

The first step is to reserve sufficient space for the array:

```
            .data
list:       .space  1000     # reserves a block of 1000 bytes
```

This yields a contiguous block of bytes of the specified size.

The label is a symbolic name for the address of the beginning of the array.




The size of the array is specified in bytes… could be used as:

- array of 1000 char values (ASCII codes)
- array of 250 int values
- array of 125 double values

There is no sense in which the size of the array is "known" by the array itself.

# Array Declaration with Initialization

An array can also be declared with a list of initializers:

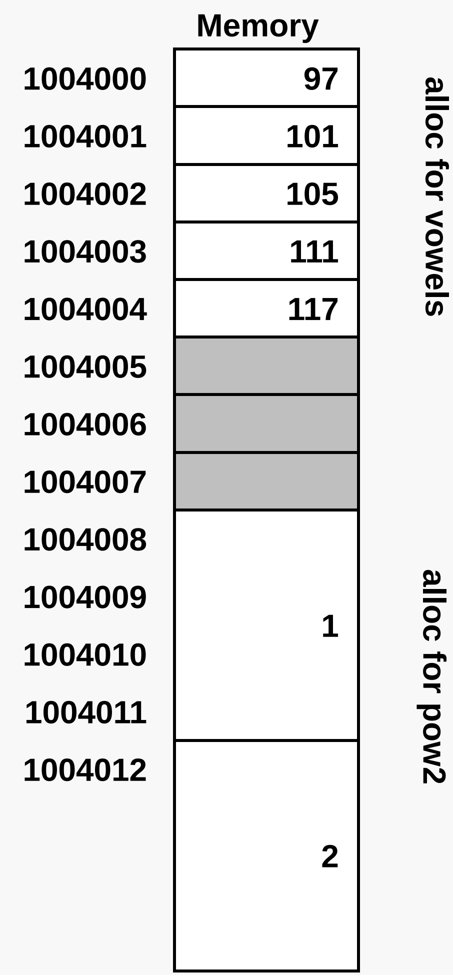
```
         .data
vowels: .byte  'a', 'e', 'i', 'o', 'u'
pow2:   .word  1, 2, 4, 8, 16, 32, 64, 128
```

`vowels` names a contiguous block of 5 bytes, set to store the given values; each value is stored in a single byte.

Address of `vowels[k] == vowels + k`

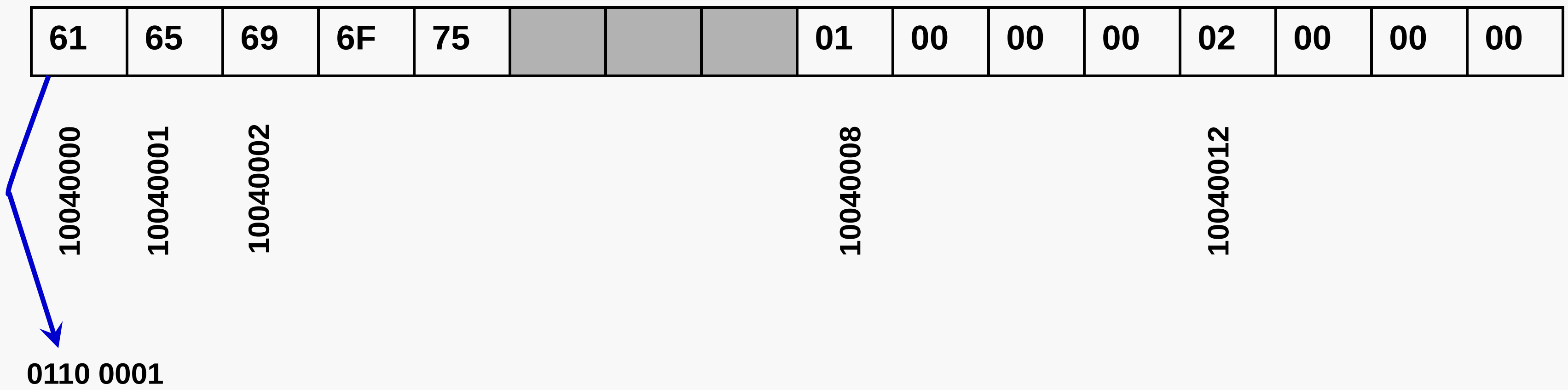
`pow2` names a contiguous block of 32 bytes, set to store the given values; each value is stored in a word (4 bytes)

Address of `pow2[k] == pow2 + 4 * k`



# Another View

Viewed as hex nybbles, the contents of memory would look like (in little-endian):




Note that endian-ness affects the ordering of bytes, not the ordering of the nybbles within a byte.

# Array Traversal and Initialization

Here's an array traversal to initialize a list of integer values:

```
        .data
list:   .space  1000
listsz: .word   250          # using as array of integers

        .text
main:   lw     $s0, listsz    # $s0 = array dimension
        la     $s1, list      # $s1 = array address
        li     $t0, 0         # $t0 = # elems init'd

initlp: beq    $t0, $s0, initdn
        sw     $s1, ($s1)     # list[i] = addr of list[i]
        addi   $s1, $s1, 4    # step to next array cell
        addi   $t0, $t0, 1    # count elem just init'd
        b      initlp
initdn:
        li     $v0, 10
        syscall
```

**QTP: why 4?**

# Array Traversal Details

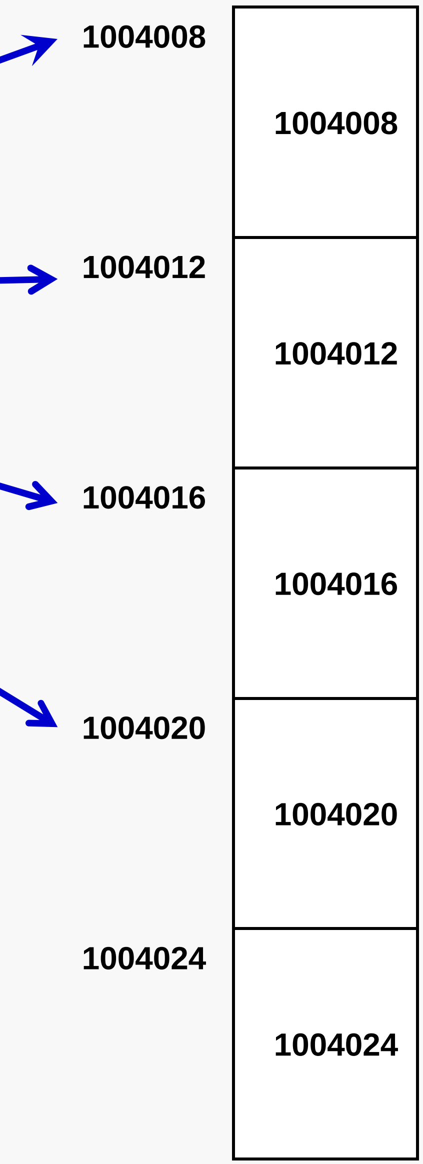
```
        . . .
initlp: beq    $t0, $s0, initdn

        sw     $s1, ($s1)

        addi   $s1, $s1, 4

        addi   $t0, $t0, 1

        b      initlp
initdn:
```




A variable that stores an address is called a *pointer*.

Here, `$s1` is a pointer to a cell of the array `list`.

We can re-target `$s1` to a different cell by adding an appropriate value to it.

# Alternate Traversal Logic

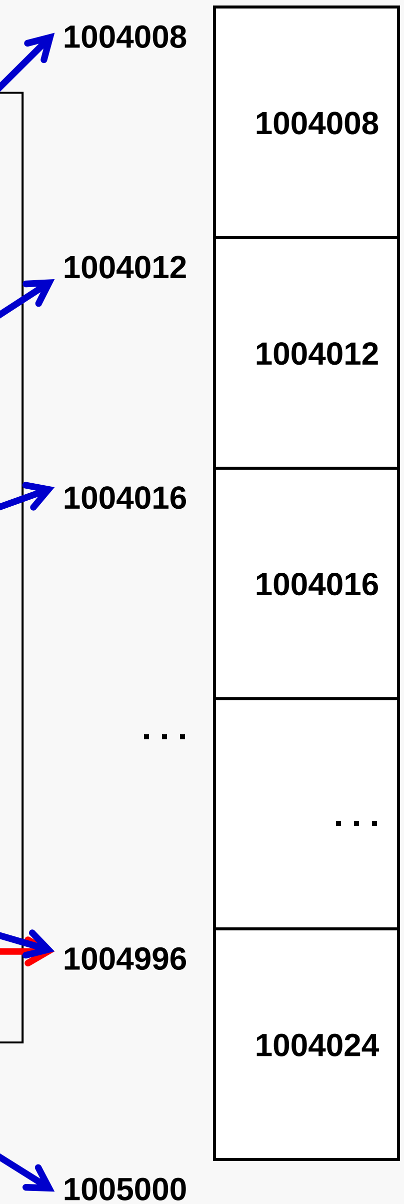


This traversal uses pointer logic to terminate the loop:

```
        .data
list:   .space  1000
listsz: .word   250

        .text
main:   la     $s1, list
        lw     $s0, listsz
        addi   $s0, $s0, -1   # index of last cell
        sll    $s0, $s0, 2    # offset of last cell
        add    $s0, $s0, $s1  # ptr to last cell

initlp: bgt    $s1, $s0, initdn
        sw     $s1, ($s1)
        addi   $s1, $s1, 4
        b      initlp
initdn:
        li     $v0, 10
        syscall
```



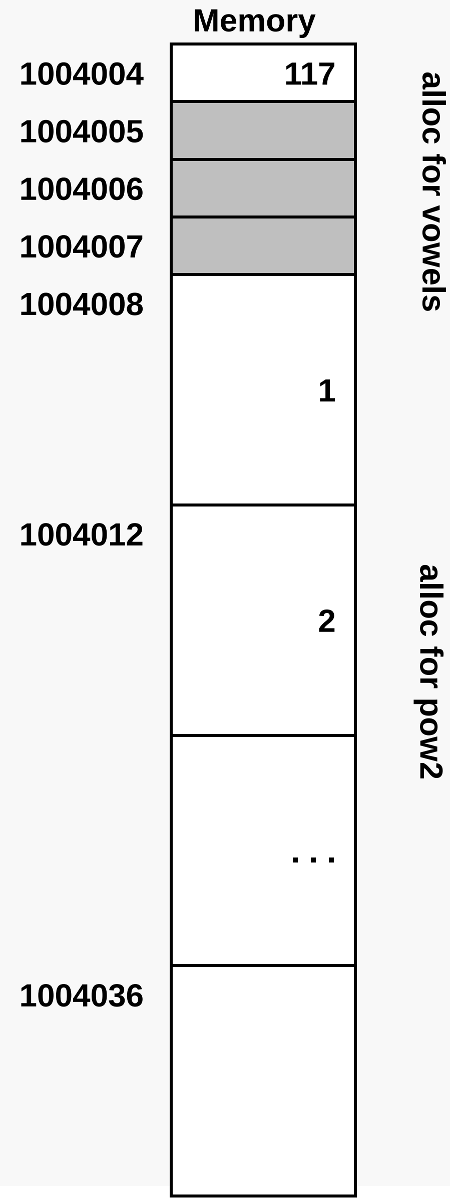

**QTP: rewrite this using the do-while pattern shown in the previous lecture**

# Array Bounds Issues



An array can also be declared with a list of initializers:

```
        .data
vowels: .byte  'a', 'e', 'i', 'o', 'u'
pow2:   .word  1, 2, 4, 8, 16, 32, 64, 128
```




What happens if you access an array with a logically-invalid array index?

`vowels[5]` ?? contents of address 1004005

While `vowels[5]` does not exist <u>logically</u> as part of the array, it does specify a physical location in memory.

What is actually stored there is, in general, unpredictable.

In any case, the value is not one that we want…

# Special Case: Array of Characters

As we've seen, the declaration:

```
        .data
vowels: .byte  'a', 'e', 'i', 'o', 'u'
```

Leads to the allocation:

| | | | | |
|---|---|---|---|---|
| 61 | 65 | 69 | 6F | 75 |

However, the declaration:

```
        .data
vowels: .asciiz  "aeiou"
```

Leads to the allocation:

| | | | | | |
|---|---|---|---|---|---|
| 61 | 65 | 69 | 6F | 75 | 00 |

An extra byte is allocated and initialized to store 0x00, which acts as a marker for the end of the character sequence (i.e., string).

This allows us to write loops to process character strings without knowing the length of the string in advance.

# Example: Searching a Character String

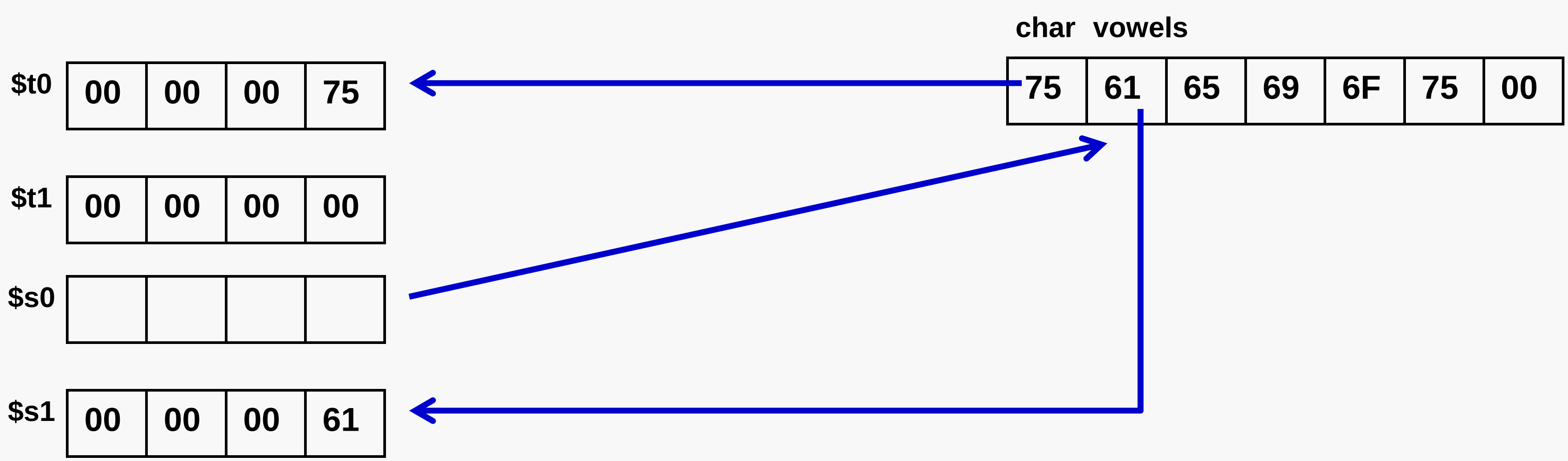
```
        .data
char:   .byte    'u'
vowels: .asciiz  "aeiou"

        .text
main:
        lb      $t0, char      # load character to look for
        li      $t1, 0         # it's not found yet
        la      $s0, vowels    # set pointer to vowels[0]
        lb      $s1, ($s0)     # get vowels[0]

srchlp: beq     $s1, $zero, srchdn   # check for terminator
        seq     $t1, $s1, $t0        # compare characters
        bgt     $t1, $zero, srchdn   # check if found
        addi    $s0, $s0, 1          # no, step to next vowel
        lb      $s1, ($s0)           # load next vowel
        b       srchlp
srchdn:
        li      $v0, 10
        syscall
```

# Example: Setup Details

```
    . . .
    lb      $t0, char       # load character to look for
    li      $t1, 0          # it's not found yet
    la      $s0, vowels     # set pointer to vowels[0]
    lb      $s1, ($s0)      # get vowels[0]
    . . .
```



# Example: Loop Details

```
        . . .
srchlp: beq     $s1, $zero, srchdn  # string terminator is 0x00

        seq     $t1, $s1, $t0       # $t1 = 1 iff $s1 == $t0

        bgt     $t1, $zero, srchdn  # if match found, exit loop

        addi    $s0, $s0, 1         # step to next elem of vowels

        lb      $s1, ($s0)          # load next elem of vowels

        b       srchlp
srchdn:
        . . .
```

# Example: Print Array Contents



```
            .data
list:       .word    2, 3, 5, 7, 11, 13, 17, 19, 23, 29
size:       .word    10
            . . .
            lw       $t3, size
            la       $t1, list          # get array address
            li       $t2, 0             # set loop counter
prnlp:
            beq      $t2, $t3, prndn    # check for array end

            lw       $a0, ($t1)         # print list element
            li       $v0, 1
            syscall

            la       $a0, NL            # print a newline
            li       $v0, 4
            syscall

            addi     $t2, $t2, 1        # advance loop counter
            addi     $t1, $t1, 4        # advance array pointer
            b        prnlp              # repeat the loop
prndn:
```

# Example: syscall Details



```
    . . .                  # syscall #1 prints and integer to stdout
    lw      $a0, ($t1)    # takes value via register $a0
    li      $v0, 1        # takes syscall # via register $v0
    syscall
    . . .
```

```
    . . .                   # syscall #4 prints asciiz to stdout
    la      $a0, NL         # takes address of string via $a0
    li      $v0, 4          # takes syscall # via register $v0
    syscall
    . . .
```

# Example: Palindromes

A *palindrome* is a sequence of characters that reads the same from left to right as from right to left:

able was i ere i saw elba

anna

madam

It is generally permitted to adjust capitalization, spaces and punctuation:

A man, a plan, a canal, Panama!

Madam, I'm Adam.

For the purpose of an example, we will not allow such manipulations.

# Example: Reading a String



We must reserve space to store the characters:

```
buffer:    .space  1025   # 1024 maximum, plus a terminator
```

We'll want to issue a prompt to the user to enter the string to be tested:

```
user_prompt:
          .asciiz   "Enter ... of no more than 1024 characters.\n"
```

We can use a couple of system calls to get the input:

```
main:
          ## Prompt the user to enter a string:
          la    $a0, user_prompt
          li    $v0, 4
          syscall
          ## Read the string, plus a terminator, into the buffer
          la    $a0, buffer
          li    $a1, 1024
          li    $v0, 8
          syscall
```

# Example: Finding the End of the String

We must locate the end of the string that the user entered:

```
        la    $t1, buffer     # lower array pointer = array base
        la    $t2, buffer     # start upper pointer at beginning

LengthLp:
        lb    $t3, ($t2)      # grab the character at upper ptr
        beqz  $t3, LengthDn   # if $t3 == 0, we're at the terminator
        addi  $t2, $t2, 1     # count the character
        b     LengthLp        # repeat the loop
LengthDn:

        addi  $t2, $t2, -2    # move upper pointer back to last char
```

**QTP: why -2?**

# Example: Testing the String



Now we'll walk the pointers toward the middle of the string, comparing characters as we go:

```
TestLp:
    bge   $t1, $t2, Yes   # if lower pointer >= upper pointer, yes

    lb    $t3, ($t1)      # grab the character at lower ptr
    lb    $t4, ($t2)      # grab the character at upper pointer

    bne   $t3, $t4, No    # if different, it's not a palindrome

    addi  $t1, $t1, 1     # increment lower ptr
    subi  $t2, $t2, 1     # decrement upper ptr

    b     TestLp          # restart the loop
```

# Example: Reporting Results



```
Yes:
    la      $a0, is_palindrome_msg     # print confirmation
    li      $v0, 4
    syscall
    b       exit

No:
    la      $a0, is_not_palindrome_msg  # print denial
    li      $v0, 4
    syscall
```