Exploration 2 - Arrays in Assembly and Writing to Memory

Writing to Memory

The following exploration is written in the context of overwriting array values. Please note that array values are stored in memory locations just like any variable. As such, all the addressing modes discussed below can be applied to overwriting memory values. This is particularly useful when you want to overwrite a memory value from within a called procedure, and all you have is the address (input/output and output parameters).

Introduction to Arrays

In **Module 7, Exploration 1 - Passing Parameters on the Stack (https://canvas.oregonstate.edu/courses/1910943/pages/exploration-1-passing-parameters-on-the-stack)** a new **addressing mode** (Base+Offset) was introduced. This was introduced to access stack contents below the topmost element, but it was also used to access content from within an **array**. You've probably been using arrays, in one form or another, since you were a child. An array is just a list that has specified positions in the list (e.g. an alphabet). In high level languages, array processing and indexing is quite straightforward. In assembly language you will need to manage the array address calculations yourself. In this exploration, you will gain some additional tools for accessing, modifying, and displaying array contents. We'll also look at how arrays are passed to (and processed in) procedures. Finally, the optional materials are strongly recommended if you plan on using pseudo-random integers.

Declaring Arrays

It's important to understand how arrays are represented in memory. We've covered this briefly already, but a deeper dive is going to be critical. When an array is declared, the **length** (number of *elements*) and the **type** (number of bytes per *element*) are both defined. The array declaration will define the memory space by determining the **size** (total number of bytes in the *array*) required by multiplying the **array length** by the **array type**. The array may or may not be initialized to specific values. You have already seen several array declarations, but here is a brief review of several formats, with explicitly identified *lengths*, *types*, and *sizes*.

```
MAXSIZE = 100
.data
list    DWORD   MAXSIZE DUP(?)  ; DUP declaration, Uninitialized values.
                                 ; length (100) x type (4) = size (400)
valArr  DWORD   100,200,0AAh,10101010b,800    ; Comma-delimited definition
```

```
myStr   BYTE    "One Byte Per Character!",0   ; length (5) x type (4) = size (20)
                                              ; Character array (string) declaration
                                              ; Includes comma-delimitation
                                              ; length (24) x size (1) = 24 (size)
```

It's vital to remember that each array is laid out in contiguous memory. A *data label* (in this case `list`, `valArr`, and `myStr`) points to some memory address, and all of the elements of the array must follow in *contiguous, increasing memory addresses* all the way to the end of the array. You can not split the array in the middle without destroying the access method that arrays use. In high level languages the same thing is true, but most of that problem is managed for you somewhere behind the scenes. In MASM when you declare an array in the data segment, the system will allocate the **size** of the array (in bytes) in contiguous cells so address calculations for specific elements can be carried out.

When declaring arrays, declare them to be *at least* big enough to handle the maximum expected length. It is a good practice to use a constant to set the array length so the length may be adjusted to accommodate different data sets. It's also a good idea to declare some kind of `arrayCount` integer to keep track of how many elements are *currently* in the array. In the simplest form of arrays, the array will be filled starting at the first position with the first element at index 0, the second element at index 1, and so forth. This type of array is called **compacted**, meaning that only the first *n elements* of the array will be in use at any given time. If the array becomes full (the number of elements equals the length), then the software engineer needs to utilize a **dynamic memory allocation mechanism** (e.g. `malloc` in C) to change size of the array. This is very difficult in assembly, so most assembly programmers will write in-line C code and use a compiler to generate the required assembly code. You won't be expected to do this in this class, so make sure your arrays are large enough when you declare them.

In a high level language, there are protections in place to prevent you from running off the end of an array (e.g. trying to access the 101st element of a 100-element array). In assembly language, you don't get any such protection. As we mentioned before, when the data is laid out in the data segment, the memory addresses are sequential in the order they were declared in. In MASM, attempting to access the 101st element of a 100-element array would be identical to accessing the four bytes declared immediately *after* that array! An entirely different variable! It is difficult to predict what could happen if you start overwriting bunches of data segment variables on accident, so be careful to keep track of your array lengths!

Data-Related Operators

These **operators** will be extremely useful when working with Arrays.

- `TYPE`
- `LENGTHOF`
- `SIZEOF`
- `OFFSET`

```
.data
  myArr     DWORD       100,200,300,400,500
  typeArr   DWORD       TYPE myArr      ; typeArr = 4 (type DWORD = 4 bytes)
  count     DWORD       LENGTHOF myArr  ; count = 5
  numByte   DWORD       SIZEOF myArr    ; numByte = 5x4=20
```

The `TYPE` operator returns the *number of bytes in the data type* used in the declaration of a given data label. Above, `myArr` is declared as an array of DWORDs, so since a DWORD is 4 bytes, `TYPE myArr` resolves to 4. `TYPE` may also be used directly with the data type (i.e. `TYPE DWORD` resolves to 4, `TYPE WORD` resolves to 2, and so on).

The `LENGTHOF` operator returns the length used in the declaration of a given data label. For Arrays, this would return the *number of elements in the array*. Above, `myArr` is declared with five comma-separated values, so `LENGTHOF myArr` returns 5. Somewhat uselessly, `LENGTHOF` may also be used with singular data labels (e.g. `LENGTHOF count` would return 1).

The `SIZEOF` operator returns the size of memory assigned in the declaration of a given data label. For Arrays, this would return the *total number of bytes in the array* (`LENGTHOF` x `TYPE` = `SIZEOF`). Above, `myArr` is declared as a 5-element DWORD array, and 5 x 4 = 20, so `SIZEOF myArr` returns 20. Entirely uselessly, `SIZEOF` may also be used with singular data labels (e.g. `SIZEOF count` would return 4).

The `OFFSET` operator is not new to you, but now you have enough information to deeply understand what it does. It returns the *address offset from the start of the data segment* of a given data label. This gives you a pointer to a memory variable. If you want to access or make any changes to an array value, or want to return a parameter from a procedure, you'll need the address in memory where it is stored. For all these situations you'll need to use the `OFFSET` operator. In fact, this is why the Irvine procedures `WriteString` and `ReadString` require you to put the `OFFSET` of the byte array into `EDX`!

Recall that arrays may be defined in multiple ways: if the type (such as BYTE) is used on each line of a multi-line definition then `LENGTHOF` and `SIZEOF` will only read the first line (see **Module 2 Exploration 3 - Defining Data in MASM x86 Assembly (https://canvas.oregonstate.edu/courses/1910943/pages/exploration-3-defining-data-in-masm-x86-assembly)**).

Check your knowledge!

1. Write a statement to declare `wArr`, an array of 150 WORDs, which does not initialize the values.
   Click here to reveal the answer.

   ```
   wArr WORD 150 DUP(?)
   ```

2. Write a statement to declares `dwArr`, an array of 76 DWORDs, which initializes the elements all to a value of 10.
   Click here to reveal the answer.

   ```
   dwArr DWORD 76 DUP(10)
   ```

3. What would `LENGTHOF dwArr` evaluate to?
   Click here to reveal the answer.

   76

4. What would `SIZEOF dwArr` evaluate to?
   Click here to reveal the answer.

   304
   76 x 4 = 304

5. What would `TYPE wArr` evaluate to?
   Click here to reveal the answer.

   2
   WORD is a 2-byte data type

Accessing Array Elements

We previously discussed **Base+Offset** addressing of array elements, but this mode is a special case of one of the two primary array access methods:

- Indirect Operands
  - Base+Offset
  - Register Indirect
- Indexed Operands

Array Indexing

Remember to index arrays from 0. The first (1^{st}) element of the array is at index 0 (list[0]), the second (2^{nd}) element of the array is at index 1, and so forth. Therefore we say the **n^{th} element** of an array is at **index (n-1)**. A general formula for array address calculation follows.

Address of `list[n]` *= (Address of* `list`*) + ((n-1) x (*`TYPE` *of element))*

For example, the address of the 10^{th} element of a DWORD `array` would be calculated as…

(Address of `array`*) + (9 x 4)*

It's important to note the inclusion of `TYPE` here. Higher level languages don't require this, so for instance in C, `array[2]` would directly access the third element of an array. This same statement, in assembly, would simply access memory starting at the third BYTE of the array. In a DWORD array, this would be halfway through the first element!

Indirect Operands

An indirect operand may be any of the 4-byte multi-purpose registers, surrounded by brackets (e.g. `[EBP]`). In general, register `ESI` is used for *source operand operations* such as pulling data from an array, and register `EDI` is used for *destination operand operations* such as overwriting an element of an array. The following code would pull the first element of `myArr` into `EAX`, and then replace the first element of `uArr` with that value.

```
.data
myArr    DWORD   100, 400, 900
uArr     DWORD   10 DUP(?)

.code
main PROC
  MOV   ESI, OFFSET myArr    ;Address of first element of myArr into ESI
  MOV   EDI, OFFSET uARR     ;Address of first element of uArr into EDI
  MOV   EAX, [ESI]           ;Value of first element of myArr into EAX
  MOV   [EDI], EAX           ;Value of EAX replaces first element of uArr
  exit
main ENDP
```

Why not do this exchange with one instruction: `MOV [EDI], [ESI]`? Because this would constitute a memory-to-memory operation, and `MOV` does not have that as a possible instruction format.

The above example shows how to access and overwrite the *first element* of the array, and we've already seen how to use **Base+Offset** addressing to access other array elements (`[EDI + n]` where *n* is the number of *bytes* to offset). This is useful for accessing a specific element on an array while maintaining the *base pointer*. NOTE that this value *n* may be an immediate or a register, so `[EDI + EAX]` is a valid **Base+Offset** reference.

Register Indirect Addressing

With **Register Indirect** addressing, the concept of a *base pointer* is abandoned and the register *itself* is incremented or decremented to change the pointer to another element of the array. This is an ideal method of stepping through arrays for printing, filling, or sometimes sorting. The following code snippet will print out each element of the DWORD array `myArr`.

```
.data
  myArr     DWORD       100,200,300,400,500

.code
main PROC
  MOV   ESI, OFFSET myArr    ;Address of first element of myArr into ESI
  MOV   ECX, LENGTHOF myArr  ;Number of elements of myArr into ECX
_PrintArr:
```

```
  MOV   EAX, [ESI]          ;n-th element of myArr into EAX
  CALL  WriteDec
  CALL  CrLf
  ADD   ESI, TYPE myArr     ;Increment ESI by 4 to
                            ;point to the next element of myArr
  LOOP _PrintArr
  exit
main ENDP
```

Indexed Operands

An **indexed operand** adds an *offset* to an *address* to generate an *effective address*. There are several different syntaxes for this and it is important to remember that the *data label* of an **array** is a placeholder for that arrays **base address**. More generally, Indexed Operands addressing combines an array *name* with a byte-distance *offset* to the element of interest.

To illustrate this, assume the code below has executed to Execution Point A, the following table shows example syntax (yes, they all work) for the various types of Indexed Operand addressing.

```
 .data
  myArr     DWORD       100,200,300,400,500

.code
main PROC
  MOV   EAX, 4
  ; Execution Point A
  exit
main ENDP
```

Reference Type	Example Instruction	Index Accessed	Value Accessed
Label[imm]	`myArr[8]`	2	300
[Label + imm]	`[myArr + 12]`	3	400
Label[reg]	`myArr[EAX]`	1	200
[Label + reg]	`[myArr + EAX]`	1	200
Label[reg * imm]	`myArr[EAX * TYPE myArr]`	4	500
[imm * reg + Label]	`[TYPE myArr * EAX + myArr]`	4	500

As you can see, the order of the computations within square brackets doesn't matter. It is also important to note the difference between higher level language syntax `myArr[8]` which would access the 9th *element*, and Assembly syntax `myArr[8]` which would access...

- The 9th *byte* of `myArr`, which is also...
- The 3rd *element* of `myArr` if declared as a DWORD array, or...
- The 5th *element* of `myArr` if declared as a WORD array.

Check your knowledge!

1. Which register is normally used for source-operand operations (operations where you are using a register-offset addressing mode to pull data from memory).
   Click here to reveal the answer.

   ESI

2. Which register is normally used for destination-operand operations (operations where you are using a register-offset addressing mode to overwrite memory).
   Click here to reveal the answer.

   EDI

3. Which addressing mode is most often used to access stack-passed parameters?
   Click here to reveal the answer.

   Base+Offset

4. Which addressing mode is most suitable for iterating through an array?
   Click here to reveal the answer.

   Register Indirect, though the others work as well.

5. Given the following .data segment declarations and assuming code has executed to Execution Point A, write statements (in the requested addressing modes) to transfer the requested values into the specified registers.

```
.data
  myArr1    WORD    123,248,300,400,500
  myArr2    DWORD   223,562,762,955,1000

.code
main PROC
  MOV   EAX, 4
  MOV   ESI, OFFSET myArr1
  MOV   EDI, OFFSET myArr2
  ; Execution Point A
  exit
main ENDP
```

   a. Use Base+Offset addressing to transfer the value 300 from `myArr1` to destination register `BX`
   Click here to reveal the answer.

   `MOV BX, [ESI+4]` or `MOV BX, [ESI+EAX]`

b. Use Base+Offset addressing to transfer the value 955 from `myArr2` to destination register `EBX`

Click here to reveal the answer.

`MOV EBX, [EDI+12]` or `MOV EBX, [EDI + 3*EAX]`

c. Use Indexed Operands addressing to transfer the value 762 from `myArr2` to destination register `ECX`

Click here to reveal the answer.

`MOV ECX, myArr2[8]` or `MOV ECX, [myArr2 + 2 * TYPE myArr2]` or ...

d. Use Indexed Operands addressing to transfer the value 400 from `myArr1` to destination register `DX`

Click here to reveal the answer.

`MOV DX, myArr1[3*TYPE myArr1]` or `MOV DX, myArr1[6]` or ...

e. Write code using Register Indirect Addressing (which would appear starting at Execution Point A) to print all values of `myArr2`, separated by line breaks, *in reverse order*.

Click here to reveal the answer.

```
  ADD   EDI, TYPE myArr2 * (LENGTHOF myArr2 - 1)    ; Point to last element of array
  MOV   ECX, LENGTHOF myArr2    ; Init loop counter = elements of array
_decPrintLoop:
  MOV   EAX, [EDI]
  CALL  WriteDec
  CALL  CrLf
  SUB   EDI, TYPE myArr2    ; Decrement by size of array type
  LOOP _decPrintLoop
```

Other code can accomplish this task.

f. (CHALLENGE) What value is in the `EAX` register after the following operation?

```
MOV EAX, [ESI + 1]
```

Click here to reveal the answer.

Hex: 2C00F800h

Decimal: 738,260,992

myArr1 in memory (hex) would appear as…

`7B 00 F8 00 2C 01 90 01 F4 01`

The line above would start accessing memory just after the 7B (the 00) and pull the next four bytes: (`00 F8 00 2C`) which are ordered little endian. Converting this to more

readable big-endian gives `2C 00 F8 00`, so the hex value stored in EAX is 2C00F800h. Converting this to decimal we get 738,260,992.

Modifying Array Values and Other Reference Parameters

Interestingly, the same basic syntax is used to modify array values as to read them. This means you can use any of the addressing modes above, but in the *destination operand* rather than the source!

- Indirect Operands
  - Base+Offset
  - Register Indirect
- Indexed Operands

The PTR Operator

Many instructions will need additional information when the destination operand is a de-referenced pointer, because a memory reference doesn't have an implicit TYPE (or size). If the source operand has an implicit size (for example, `EAX` is 4 bytes), the assembler can infer that it must write (or modify) 4 bytes. However, when the source operand also does not have an implicit size (as in the example code snippet below), there's not sufficient information for the assembler to select an op-code.

```
  MOV   EDI, OFFSET myVar  ; EDI points to a memory variable
  MOV   [EDI], EAX         ; EAX is size 4, so assembler knows to use opcode corresponding to
a 4-byte MOV
  MOV   [EDI], 10          ; Immediate 10 has no known size
  INC   [EDI]              ; INC can work on 1-, 2-, 4-byte lengths. Which to use?
```

In the second line of the code snippet above, `EAX` has a known size of 4 bytes, so this operation is legal. However in the third and fourth lines, there is not enough information for the assembler to know how many bytes will be transferred, so both of those instructions will fail. The `MOV [EDI], 10` will give an “invalid instruction operands” error and `INC [EDI]` will give an “instruction operand must have size” error.

Fortunately MASM has a fix for this! The fix is another useful **operator**: `PTR` . `EDI`, in the code above, points to a DWORD variable (4 bytes). The `PTR` operator allows us to *explicitly specify the number of bytes* to write to memory! Syntax is simple. Before the memory reference, insert `data_type PTR` where `data_type` is the data type you’re casting to… in general this will be WORD, DWORD, etc… Let’s fix that code snippet!

```
  MOV   EDI, OFFSET myVar     ; EDI points to a memory variable
  MOV   [EDI], EAX            ; No change needed
  MOV   DWORD PTR [EDI], 10   ; Cast immediate 10 as DWORD, then write to memory
  INC   DWORD PTR [EDI]       ; Increment 4-byte unsigned integer in memory
                              ; pointed to by EDI
```

Arrays in Procedures: Passing, Accessing, Modifying

In order to access/modify an array from within a subprocedure you need to be able to pass it as a parameter. It's a bad idea to try to pass any array (ever) by value because you'd have to push every single array element, which is a short trip to Stack Overflow! Even if you did manage to get all array elements on the stack, they'd be passed as value parameters and so you wouldn't be able to modify the array in memory! For this reason you will always pass arrays (and other data structures) by reference! In fact, you've already seen this when you passed a string address (`PUSH OFFSET name`).

Suppose that some procedure (arrayFill) fills an array with 32-bit integers. The calling program passes the address of the array, along with a count of the array length.

```
COUNT = 100
.data
  list DWORD COUNT DUP(?)

.code
main PROC
  ;...
  PUSH OFFSET list  ; Pass address of first element in list
  PUSH COUNT
  CALL arrayFill
  ;...
  exit
main ENDP
```

So passing the parameter is just as we have done before for other arrays (without knowing why we were doing it, perhaps). We've made an entire array a return parameter, and only had to put four bytes on the stack to do so! Speaking of the stack, what does it look like right after `CALL arrayFill`?

- Return Address (on top of the stack)
- Number of elements to be filled
- Address of the first element of *list*

Moving on to the actual procedure body, the address of *list* and the *count* must be copied off the stack and saved into registers for processing. This is nothing new, but is included here for accuracy.

```
arrayFill PROC
  PUSH  EBP             ; Build Stack Frame
  MOV   EBP, ESP
  PUSH  EAX             ; Preserve used registers
  PUSH  ECX
  PUSH  EDI

  MOV   ECX, [EBP+8]    ; List length into ECX
  MOV   EDI, [EBP+12]   ; Address of list into EDI

  ; Code Block A
```

```
  POP   EDI
  POP   ECX
  POP   EAX             ; Restore used registers
  POP   EBP
  RET   8
arrayFill ENDP
```

Now we have a framework to work in! There are a few methods which might be used here. Obviously *indexed operands addressing* is not going to work because that requires using global variables, and that's poor practice and breaks our carefully-constructed modularization. So let's write Code Block A with both Register Indirect and Base+Offset addressing modes!

Register Indirect

With *Register Indirect*, store new values into the location pointed to by `EDI` (with proper de-referencing, of course) then increment `EDI` to point to the next array element. Since *list* is a DWORD array, `EDI` must be incremented by 4. Fortunately we don't need the `PTR` operator because the source operand (`EAX`) implies a 4-byte transfer.

This code will step through the loop, overwriting one value per iteration. The actual code to generate the values is left out because it's not relevant to the discussion here.

```
  ; Code Block A (Register Indirect)
_fillLoop:
  ; ...code to generate some value and store it in EAX goes here...
  MOV   [EDI], EAX  ; Overwrite value in memory pointed to by EDI.
  ADD   EDI, 4      ;Increment pointer by type size, point to next value.
  LOOP  _fillLoop
```

This same general structure will work for writing any values to memory, and is generally the preferred method for writing *output parameters*.

Base+Offset

The key in *Base+Offset* is that the base pointer to the array (in this case EDI) remains unchanged. This means we'll need another register to increment!

```
  ; Code Block A (Base+Offset)
  PUSH  EBX
  MOV   EBX, 0    ;Zero initial offset (alternately: XOR EBX, EBX)
_fillLoop:
  ; ...code to generate some value and store it in EAX goes here...
  MOV   [EDI+EBX], EAX  ; Base EDI + Offset EBX
  ADD   EBX, 4          ; Offset increments by type size.
  LOOP  _fillLoop
  POP EBX
```

Clearly this is a bit more complicated, but it does preserve the array base pointer, in case you need that somewhere else in the procedure.

Example

Because it can be difficult to implement these functions as a beginner, this **Example of Array Filling and Printing (https://canvas.oregonstate.edu/courses/1910943/files/94857672/download?wrap=1)** ↓ (https://canvas.oregonstate.edu/courses/1910943/files/94857672/download?download_frd=1) is being provided. It is recommended you study this program, run it, tweak it, and make sure you understand how and why it works.

Check your knowledge!

1. (True/False) The instruction `INC [EDI]` is valid.
   Click here to reveal the answer.

   False
   `[EDI]` does not have an implied size, so this will cause an error.

2. (True/False) The instruction `ADD DWORD PTR [EDI+3], 20` is valid.
   Click here to reveal the answer.

   True
   add mem, imm is an allowed instruction format.

3. Given the following partial listing file:

```
        MAX = 50
        .data
        ;...
0300    list DWORD       MAX DUP(0)
????    a    DWORD       25
????    b    DWORD       15
        ;...

0000    main        PROC
0000      push  a
0005      push  b
000A      push  OFFSET list
000F      call  someProc
0014      ; More Code ...

          exit      ;exit to operating system
006C    main        ENDP

006C    someProc    PROC
006C      push  ebp
006F      mov   ebp, esp
0072      ; Execution Point A, more code...

008B      pop   ebp
008E      ret   12    ;return to calling procedure
008F    someProc    ENDP
```

Assume the address of *list* is 0300h as seen above, and the initial values (at the top of *main* before executing any instructions) are `ESP` = 0A04h, `EBP`= 0BB8h. *main* has called *someProc*, and the first two statements of *someProc* have been executed, so execution is paused at Execution Point A.

a. What is the (hexadecimal) address of *a*?
Click here to reveal the answer.

03C8h
list takes 200 (decimal) bytes of memory = 0xC8 bytes.
So the address of a is the address of *list* + the size of *list* = 0300h + 0C8h = 03C8h

b. What is the (hexadecimal) address of the 33rd element of *list* ?
(Hint: in Python or Java, the 33rd element is referenced by `list[32]`)
Click here to reveal the answer.

0380h
32 elements of *list* take 32 x 4 = 128 (decimal) bytes of memory = 80h bytes
So the address of the 33rd element is the address of *list* + the number of bytes taken by the first 32 elements = 0300h + 80h = 0380h

c. At *Execution Point A*, show the contents of the runtime stack, formatted like this:

Address	Contents	Meaning
09E8		
09EC		
09F0		
09F4		
09F8		
09FC		
0A00		
0A04	?	unknown

Click here to reveal the answer.

See table:

Address	Contents	Meaning
09E8		
09EC		

Address	Contents	Meaning
09F0	0BB8h	old EBP value
09F4	0014h	return address from someProc
09F8	0300h	address of *list*
09FC	15	Value of b when PUSHed
0A00	25	Value of a when PUSHed
0A04	?	unknown

d. At *Execution Point A*, what is the value the `EBP` register?
Click here to reveal the answer.

09F0h
Value of ESP just after EBP was pushed, calculated as 0A04h - 4h (a value) - 4h (b value) - 4h (*list* address) - 4h (return address) - 4h (old value of EBP) = 0A04h - 0014h = 09F0h

e. At *Execution Point A*, write code to move the value of the *b* th element of *list* into the `EBX` register. (Consider b=0 to be the 1st element of *list*). Global labels *b* and *list* are not permitted.
Click here to reveal the answer.

```
MOV ESI, [EBP + 8]    ; move the OFFSET of list into ESI
MOV EAX, 4            ; there are 4 bytes per DWORD
MOV EBX, [EBP + 12]   ; move the value of b into EBX
MUL EBX               ; Multiply EAX by b to (almost) get
                      ; the offset to the b-th element
SUB EAX, 4            ; EAX now holds the offset from ESI to
                      ; the bth element of list
MOV EBX, [ESI+EAX]    ; move the element into EBX
```

Optional Additional Resources

- **Array Filling / Printing Example (https://canvas.oregonstate.edu/courses/1910943/files/94857672/download?wrap=1)** (https://canvas.oregonstate.edu/courses/1910943/files/94857672/download?download_frd=1) - Example array manipulation program using procedures with parameters passed on the runtime stack.
- **Irvine Procedures Reference (https://canvas.oregonstate.edu/courses/1910943/files/94857677?wrap=1)** - Check out the `Randomize` and `RandomRange` procedures!

See if you can use these procedures to supplement the *arrayFill* procedure from the sample code above to fill the array with random integers!