## Storing and Accessing Arrays

So far, we have assumed that variables stored in memory contain scalar values

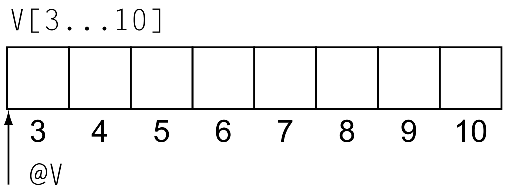
- many programs need arrays or similar structures
- locating and referencing an element of an array can be **surprisingly complex**

We will begin by looking at **one-dimensional** arrays, *i.e.*, vectors, and then generalize the approach to **multi-dimensional** arrays.

## Referencing a Vector Element

One-dimensional arrays, or vectors, are typically stored in **contiguous memory**, so that the $i^{th}$ element immediately precedes the $i + 1^{st}$ element.

A vector `V[3...10]` leads to the following memory layout, where the number below a cell indicates its index in the vector.



When the compiler encounters a reference, it must use the **index** into the vector, along with **facts** available from the declaration of `V`, to generate an offset for the reference.

The actual address is then computed as the **sum** of the offset and a pointer to the start of `V`, which we write as `@V`.

## Referencing a Vector Element

Assume that a vector `V` has been declared as `V[`*low...high*`]`, where *low* and *high* are the vector's lower and upper bounds.

To translate the reference `V[i]`, the compiler needs both a pointer to the start of storage for `V` and the offset of element `i` within `V`.

The offset is simply $(\mathtt{i} - \mathit{low}) \times w$, where $w$ is the length of a single element of `V`.

**Example** If $w$ is $4$, the offset of `V[6]` in `V[3...10]` is: $(6 - 3) \times 4 = 12$.

## Referencing a Vector Element

Assuming that $r_i$ holds the value of i, the following code fragment computes the address of V[i] into $r_3$ and loads its value into $r_V$.

```
loadI  @V          ⇒  $r_{@V}$     ...get V's address
subI   $r_i$, 3    ⇒  $r_1$        ...offset − lower bound
multI  $r_1$, 4    ⇒  $r_2$        ...× element length (4)
add    $r_{@V}$, $r_2$ ⇒ $r_3$     ...address of V[i]
load   $r_3$       ⇒  $r_V$        ...value of V[i]
```

Notice that the simple reference V[i] introduces **three** arithmetic operations. The compiler can improve this sequence.

## Referencing a Vector Element

If *w* is a power of two, the multiply can be replaced with an **arithmetic shift**. Many base types in real programming languages have this property.

Even though adding the address and offset is unavoidable, most processors include an **addressing mode** specifically designed for this use case.

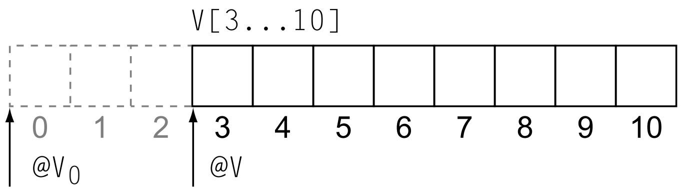
| | | | | |
|---|---|---|---|---|
| `loadI` | `@V` | $\Rightarrow$ | $\mathtt{r}_{\mathtt{@V}}$ | ...get V's address |
| `subI` | $\mathtt{r}_{\mathtt{i}}$, `3` | $\Rightarrow$ | $\mathtt{r}_{1}$ | ...offset $-$ lower bound |
| `lshiftI` | $\mathtt{r}_{1}$, `2` | $\Rightarrow$ | $\mathtt{r}_{2}$ | ...$\times$ element length $(4)$ |
| `loadAO` | $\mathtt{r}_{\mathtt{@V}}$, $\mathtt{r}_{2}$ | $\Rightarrow$ | $\mathtt{r}_{\mathtt{V}}$ | ...value of `V[i]` |

**Example** In the Intel x86-64 ISA, move has an **offset-scaled-base-relative** addressing mode. The instruction `movq -16(%rbx, %rcx, 8), %rax` loads the value at the address $-16 + \texttt{\%rbx} + \texttt{\%rcx} \times 8$ into register `%rax`.

## Referencing a Vector Element

### Using a lower bound of zero eliminates the subtraction

- if the compiler knows the lower bound of `V`, it can fold the subtraction into `@V`
- instead of using `@V` as the base address for `V`, it can also use $V_0 = @V - low \times w$



**False Zero**

The **false zero** of a vector `V` is the address where `V[0]` would be. In multiple dimensions, it is the location of a zero in each dimension.

## Referencing a Vector Element

Using $\texttt{@V}_0$ and assuming that `i` is in $\texttt{r}_\texttt{i}$, the code for accessing `V[i]` becomes shorter and, presumably, faster.

| | | | | |
|---|---|---|---|---|
| `loadI` | $\texttt{@V}_0$ | $\Rightarrow$ | $\texttt{r}_{\texttt{@V}_0}$ | ...adjusted address of `V` |
| `lshiftI` | $\texttt{r}_\texttt{i}$, `2` | $\Rightarrow$ | $\texttt{r}_1$ | ...$\times$ element length $(4)$ |
| `loadAO` | $\texttt{r}_{\texttt{@V}_0}$, $\texttt{r}_1$ | $\Rightarrow$ | $\texttt{r}_\texttt{V}$ | ...value of `V[i]` |

**Note** In a compiler, the longer sequence may produce better results by exposing details such as the multiply and add instruction to optimization.

An alternative strategy, employed in languages like C or Java, forces the use of zero as a lower bound, which ensures that $\texttt{@V}_0 = \texttt{@V}$ and simplifies all array-address calculations.

## Array Storage Layout

Accessing an element of a multi-dimensional array requires more work. The code that the compiler needs to generate depends on the mapping of array indices to memory locations.

**Most implementations use one of three schemes**

- row-major order
- column-major order
- indirection vectors

The source-language definition usually specifies one of these mappings.

## Array Storage Layout

**Example** Consider the array `A[1...2,1...4]`. Conceptually, it looks as follows.

| | | | | |
|---|---|---|---|---|
| A | 1,1 | 1,2 | 1,3 | 1,4 |
| | 2,1 | 2,2 | 2,3 | 2,4 |

In **row-major order**, the elements of a are mapped onto consecutive memory locations so that adjacent elements of a single row occupy consecutive memory locations.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1,1 | 1,2 | 1,3 | 1,4 | 2,1 | 2,2 | 2,3 | 2,4 |

## Array Storage Layout

The array storage layout has an impact on **memory access patterns** and, thus, run-time performance.

**Example** Consider the following nested loops.

```
for i ← 1 to 2
    for j ← 1 to 4
        A[i,j] ← A[i,j] + 1
```

In row-major order, the assignment statement steps through memory in **sequential order**, beginning with `A[1,1]`, `A[1,2]`, `A[1,3]`, and on through `A[2,4]`.

Switching the two loops produces an access pattern that jumps between rows. For arrays larger than the cache, lack of sequential access can lead to poor run-time performance.

## Array Storage Layout

The obvious alternative to row-major order is **column-major order**. It keeps the columns of a in contiguous locations, producing the following layout.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1,1 | 2,1 | 1,2 | 2,2 | 1,3 | 2,3 | 1,4 | 2,4 |

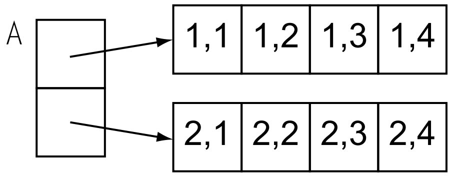
While row-major order leads to sequential access if the **rightmost** subscript varies fastest, column-major order leads to sequential access if the **leftmost** subscript varies fastest.

### Row-major vs. column-major order

For languages that store arrays in contiguous storage, row-major order has been the typical choice. The one notable exception is FORTRAN, which uses column-major order.

## Array Storage Layout

A third alternative are **indirection vectors**. This scheme reduces all multi-dimensional arrays to a set of (one-dimensional) vectors.



Indirection vectors appear simple, but they introduce their own complexity

- need more storage than either of the contiguous storage schemes
- require that application initializes all of the indirection pointers at runtime

An advantage of indirection vectors is that they can easily represent **ragged arrays**.

**Note** Java supports indirection vectors.

## Referencing an Array Element

Programs that use arrays typically contain **references** to individual array elements.

As with vectors, the compiler must translate an array reference into a base address for the array's storage and an offset where the element is located relative to the starting address.

We will look at the following cases

- row-major order
- indirection vectors
- array-valued parameters

**Note** The calculations for column-major order follow the same basic scheme as those for row-major order, with the dimensions reversed.

## Row-Major Order

Before deriving the corresponding formula, we introduce the following notation

- $low_\mathtt{i}$ and $high_\mathtt{i}$ denote *low* and *high* of the `i`-th dimension, respectively
- $len_\mathtt{i} = high_\mathtt{i} - low_\mathtt{i} + 1$ denotes the length of the `i`-th dimension

In row-major order, the address calculation must find the start of the row and then generate an offset within the row as if it were a vector.

To access element `A[i,j]` of a two-dimensional array the compiler therefore must emit code that computes the…

- address of row `i`: $(\mathtt{i} - low_1) \times len_2 \times w$
- offset of element `j`: $(\mathtt{j} - low_2) \times w$

Putting all parts together, the resulting address computation is as follows.

$$@\mathrm{A} + (\mathtt{i} - low_1) \times len_2 \times w + (\mathtt{j} - low_2) \times w$$

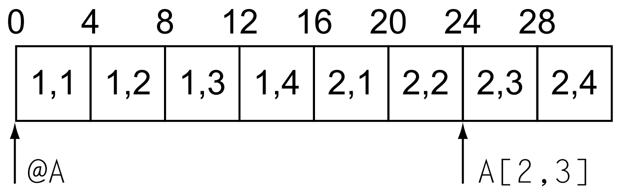
## Row-Major Order

**Example** In the array `A[1...2,1...4]`, element `A[2,3]` lies at offset

$$(2-1) \times (4-1+1) \times 4 + (3-1) \times 4 = 24$$

from `A[1,1]`, assuming that `@A` points at `A[1,1]` at offset $0$.

Looking at `A` in memory, we find that the address of `A[1,1]` $+ 24$ is, in fact, the address of `A[2,3]`.



## Row-Major Order

As in the vector case, we can simplify the address calculation if upper and lower bounds are known at compile time by introducing a false zero.

$$@A + (\mathtt{i} - low_1) \times len_2 \times w + (\mathtt{j} - low_2) \times w$$
$$@A + (\mathtt{i} \times len_2 \times w) - (low_1 \times len_2 \times w) + (\mathtt{j} \times w) - (low_2 \times w)$$
$$@A + (\mathtt{i} \times len_2 \times w) + (\mathtt{j} \times w) - (low_1 \times len_2 \times w + low_2 \times w)$$

The term $(low_1 \times len_2 \times w + low_2 \times w)$ is independent of `i` and `j`. Thus, it can be factored directly into the base address: $@A_0 = @A - (low_1 \times len_2 \times w + low_2 \times w) = @A - 20$.

Now, the array reference is simply $@A_0 + \mathtt{i} \times len_2 \times w + \mathtt{j} \times w$. Finally, we can refactor and move $w$ outside, saving an extraneous multiplication.

$$@A_0 + (\mathtt{i} \times len_2 + \mathtt{j}) \times w$$

## Row-Major Order

Assuming that `i` and `j` are in $r_i$ and $r_j$, and that $len_2$ is a constant, this form of the polynomial leads to the following code sequence.

| | | | | |
|---|---|---|---|---|
| loadI | $@A_0$ | $\Rightarrow$ | $r_{@A_0}$ | ...adjusted base address of `A` |
| multI | $r_i$, $len_2$ | $\Rightarrow$ | $r_1$ | ...`i` $\times len_2$ |
| add | $r_1$, $r_j$ | $\Rightarrow$ | $r_2$ | ...$+$ `j` |
| multI | $r_2$, 4 | $\Rightarrow$ | $r_3$ | ...$\times$ element length $(4)$ |
| loadAO | $r_{@A_0}$, $r_3$ | $\Rightarrow$ | $r_a$ | ...value of `A[i,j]` |

**Note** The computation takes two multiplications and two additions (one in the `loadAO`). The second multiply can be rewritten as a shift.

## Row-Major Order

If the compiler does not have access to the array bounds

- compute the false zero at runtime
- use the more complex polynomial with subtractions to adjust for lower bounds

The first option only makes sense if the elements of the array are accessed multiple times in a procedure.

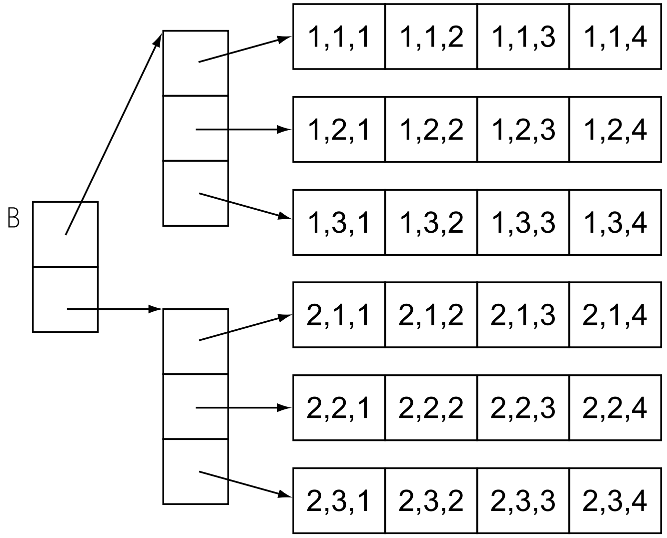
**Note** These ideas behind address computation for two-dimensional arrays generalize to **arrays of higher dimension**. The address polynomial for an array stored in **column-major order** can be derived in a similar fashion.

## Indirection Vectors

Using indirection vectors simplifies the code generated to access an individual element. For an $n$-dimensional array, the first $n - 1$ steps navigate the indirection vectors, while the last step performs a regular (one-dimensional) vector lookup.

**Example** Accessing element `B[i,j,k]` in the array `B[1...2,1...3,1...4]`

1. use `@B0`, `i`, and the length of a pointer, to find the vector for the subarray `B[i,*,*]`
2. use result, along with `j` and the length of a pointer to find the vector for the subarray `B[i,j,*]`
3. use base address in the vector-address computation with `k` and element length *w* to find the address of `B[i,j,k]`



## Indirection Vectors

If the values for `i`, `j`, and `k` exist in registers $r_i$, $r_j$, and $r_k$, respectively, and $@B_0$ is the zero-adjusted address of the first dimension, then `B[i,j,k]` can be loaded as follows.

```
loadI   @B₀       ⇒ r@B₀   ...adjusted base address of B
multI   rᵢ, 4     ⇒ r₁     ...pointer size (4)
loadAO  r@B₀, r₁  ⇒ r₂     ...load @B[i,*,*]

multI   rⱼ, 4     ⇒ r₃     ...pointer size (4)
loadAO  r₂, r₃    ⇒ r₄     ...load @B[i,j,*]

multI   rₖ, 4     ⇒ r₅     ...element size (4)
loadAO  r₄, r₅    ⇒ r_b    ...value of B[i,j,k]
```

**Note** This code assumes that the pointers in the indirection structure have already been adjusted to account for non-zero lower bounds.

## Accessing Array-Valued Parameters

**Note** Even in languages that use call by value for all other parameters, arrays are usually passed by reference, for the sake of runtime performance.

To generate array references in the callee, we need information about the dimensions of the array that is bound to the parameter.

- some languages, *e.g.*, FORTRAN, Oberon-0, give the **programmer** responsibility for passing the information to the callee needed to address a parameter array correctly
- other languages leave the task of collecting, organizing, and passing the necessary information to the **compiler**

## Accessing Array-Valued Parameters

**Dope Vector**

A descriptor for an actual parameter array that contains both a pointer to the **start of the array** and the **necessary information for each dimension**. Dope vectors may also be used for arrays whose bounds are determined at runtime.

**Caller**

- builds the dope vector and stores it in callee's AR
- value passed in the array's parameter slot is a pointer to the dope vector

**Callee**

- loads values out of the dope vector as needed
- generates array reference using the same polynomial as for a local array

## Range Checking

A program that references an out-of-bounds array element is not well formed

- some languages, *e.g.*, Ada, Java, require such accesses to be detected and reported
- compilers for other languages may include optional mechanisms for range checking

The simplest implementation of **range checking** inserts a check before each array access

- introduces additional overhead, which may be significant in some programs
- may need more information in dope vector, *e.g.*, explicit bounds rather than length

An optimizing compiler can improve on this naive implementation in several ways

- combine checks
- move checks out of loops
- prove checks redundant

## Character Strings

The operations that programming languages provide for **character data** are different from those provided for numerical data.

Programming language support for character strings varies

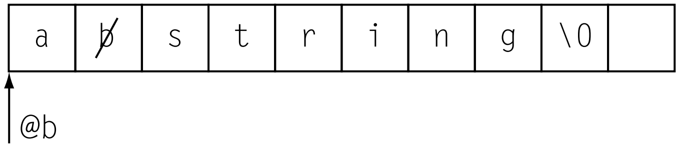
- in C, most manipulation takes the form of calls to library routines
- PL/I provides first-class mechanisms to assign individual characters, specify arbitrary substrings, and concatenate strings

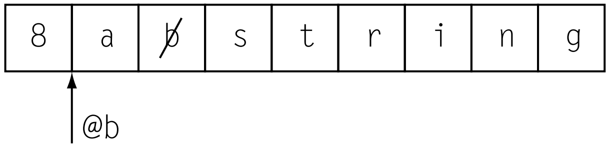
CISC architectures provide extensive support for string manipulation, whereas RISC machines rely on the compiler to code these operations using a set of simpler instructions.

- string assignment
- string concatenation
- string-length computation

## String Representation

The string representation chosen by the compiler writer has a strong impact on the cost of string operations.





- C traditionally uses **null termination** (left) with a designated character (`\0`) serving as a terminator
- many other language implementations use an **explicit length field** (right) that stores the length of the string ($8$) alongside its contents

If the length field takes more space than the null terminator, then storing the length will marginally increase the size of the string in memory. However, storing the length simplifies several operations on strings.

## String Assignment

String assignment is conceptually simple. The complexity of its implementation depends on whether or not the target machine supports **character-sized memory operations**.

**Example** In C, an assignment from the third character of b to the second character of a can be written as `a[1] = b[2];`.

With support for character-sized memory operations, this assignment translates into the following simple code.

```
loadI     @b          ⇒  r_@b
cloadAI   r_@b, 2     ⇒  r_1
loadI     r_@a        ⇒  r_@a
cstoreAI  r_1         ⇒  r_@a, 1
```

Without hardware support for character-sized memory operations, the compiler must generate more complex code that **masks** the characters explicitly.

## String Assignment

**Assumptions** Strings a and b begin on word boundaries, a character occupies 1 byte, and that a word is $4$ bytes.

| | | | | |
|---|---|---|---|---|
| `loadI` | `0x0000FF00` | $\Rightarrow$ | $\mathtt{r}_{\mathtt{C2}}$ | …mask for the character 2 |
| `loadI` | `0xFF00FFFF` | $\Rightarrow$ | $\mathtt{r}_{\mathtt{C124}}$ | …mask for characters 1, 2, and 4 |
| `loadI` | `@b` | $\Rightarrow$ | $\mathtt{r}_{\mathtt{@b}}$ | …load address of b |
| `load` | $\mathtt{r}_{\mathtt{@b}}$ | $\Rightarrow$ | $\mathtt{r}_{1}$ | …load first word of b |
| `and` | $\mathtt{r1},\ \mathtt{r}_{\mathtt{C2}}$ | $\Rightarrow$ | $\mathtt{r}_{2}$ | …mask away other characters |
| `lshiftI` | $\mathtt{r}_{2},\ 8$ | $\Rightarrow$ | $\mathtt{r}_{3}$ | …move character to first byte |
| `loadI` | `@a` | $\Rightarrow$ | $\mathtt{r}_{\mathtt{@a}}$ | …load address of a |
| `load` | $\mathtt{r}_{\mathtt{@a}}$ | $\Rightarrow$ | $\mathtt{r}_{4}$ | …load first word of a |
| `and` | $\mathtt{r}_{4},\ \mathtt{r}_{\mathtt{C124}}$ | $\Rightarrow$ | $\mathtt{r}_{5}$ | …mask away second character |
| `or` | $\mathtt{r}_{3},\ \mathtt{r}_{5}$ | $\Rightarrow$ | $\mathtt{r}_{6}$ | …insert new second character |
| `store` | $\mathtt{r}_{6}$ | $\Rightarrow$ | $\mathtt{r}_{\mathtt{@a}}$ | …store result |

## String Assignment

The code is similar for longer strings. The following code for `a = b;` assumes explicit length representation and hardware support for character-sized memory operations.

```
     loadI     @b        ⇒  r@b         ...load address of b
     loadAI    r@b, -4   ⇒  r1          ...load b's length
     loadI     @a        ⇒  r@a         ...load address of a
     loadAI    r@a, -4   ⇒  r2          ...load a's length
     cmp_LT    r2, r1    ⇒  r3          ...does b fit into a?
     cbr       r3        →  LSOV, L1    ...raise string overflow
L1:  loadI     0         ⇒  r4          ...initialize counter
     cmp_LT    r4, r1    ⇒  r5          ...more to copy?
     cbr       r5        →  L2, L3      ...jump
L2:  cloadAO   r@b, r4   ⇒  r6          ...load character from b
     cstoreAO  r6        ⇒  r@a, r4     ...store character in a
     addI      r4, 1     ⇒  r4          ...increment counter
     cmp_LT    r4, r1    ⇒  r7          ...more to copy?
     cbr       r7        →  L2, L3      ...jump
L3:  storeAI   r1        ⇒  r@a, -4     ...set length of a
```

## String Assignment

In C, which uses null termination for strings, the same assignment would be written as a character-copying loop.

```
t1 = a;
t2 = b;
do {
    *t1++ = *t2++;
} while (*t2 != '\0')
```

```
      loadI   @b        ⇒ r@b        ...load address of b
      loadI   @a        ⇒ r@a        ...load address of a
      loadI   NULL      ⇒ r1         ...load terminator
L1:   cload   r@b       ⇒ r2         ...load character
      cstore  r2        ⇒ r@a        ...store character
      addI    r@b, 1    ⇒ r@b        ...move pointer
      addI    r@a, 1    ⇒ r@a        ...move pointer
      cmp_NE  r1, r2    ⇒ r3         ...more to copy?
      cbr     r3        → L1, L2     ...jump
L2:   nop                            ...next statement
```

## String Assignment

### Further Possibilities

- if the target machine supports **auto-increment** on load and store operations, the two adds in the loop can be performed in the cload and cstore operations
- the compiler could generate **whole-word** loads and stores, followed by a character-oriented loop to handle any leftover characters at the end of the string
- the compiler might also call a carefully optimized **library routine** to implement the nontrivial cases

## String Concatenation

Concatenation is simply a shorthand for a sequence of one or more assignments. It comes in **two basic forms**.

1. Appending string b to string a
   - compiler emits code to determine the length of a
   - space permitting, it assigns b to the space that immediately follows a
2. Creating a new string that contains a followed immediately by b
   - requires copying each character in a and each character in b
   - can be treated as a pair of assignments in code generation

The compiler should ensure that **enough space** is allocated to hold the result. If the lengths of a and b are unknown at compile time, runtime code needs to be generated

- to compute the lengths of the strings
- to perform corresponding test and branch

## String Length

Programs that manipulate strings often need to compute a character string’s length.

Different string representations lead to different costs for the length computation

- a **null-terminated string** requires time proportional to the length of the string
- with an **explicit length field** the cost is constant and small

**Tradeoff** Null termination saves a small amount of space, but requires more code and more time for the length computation. An explicit length field costs one more word per string, but makes the length computation take constant time.

**Example** With explicit length fields a statement `length(a + b)` can be optimized to two `loads` and one `add`.

## String Length on Intel x86-64

The Intel x86-64 ISA provides several **dedicated operations** for string manipulation.

- `movs`, `lods`, `stos`: move, load, and store strings
- `cmps`: compare strings
- `scas`: scan strings

These instructions are combined with **repetition prefixes**: `rep`, `repe`, `repne`, `repz`, and `repnz`).

```
movq   -1, %rcx      # "clear" counting register
movq   %rdi, %rsi    # backup %rdi
mov    0, %al        # look for \0
repne scasb          # actually do the search
subq   %rsi, %rdi    # save the string length
decq   %rdi          # do not count the \0 in the string length
movq   %rdi, %rax    # save the return value
```