CS453 Arrays
in Java
in general

Plan for the day
- Arrays in general
- Array descriptors
- Type checking for arrays
- Code gen for arrays
Arrays

An array is a collection of items of the same type
- so that the address of an element can be computed from
  the start address and the index (efficiency)
- index: int (or int derivative type like unsigned or byte)

Once an array is allocated, the sizes of its dimensions do not change (as opposed to ArrayLists, Lists, …)

Java arrays are one Dimensional
- higher dimensional arrays are arrays of sub-arrays
  these are sometimes called “ragged” arrays, as the lengths of the sub-arrays can differ
- as opposed to rectangular arrays in Fortran and C
# Array representations

1. Store length with array elements
   e.g. at the front of the array
      - this is nice for Java arrays
   the array is now represented by its start address
      - the address of the length field
   when allocating and indexing in such arrays this length field must be taken into account (added / skipped over)

2. Have a separate array descriptor with
   - index ranges for all dimensions
   - widths in bytes in each dimension
   (some representation of) start address
Array descriptors

Sometimes called “dope vectors”
- dope: slang for “the essential information”

The descriptor representation is useful for rectangular arrays, especially when the language allows the creation of sub arrays from arrays.

These sub arrays can be

- lower dimensional, e.g. vector (row, column) out of matrix, or plane out of cube
- same dimensional, sometimes called windows or tiles
  e.g. median filter:
  for all (sliding) windows of 3x3 in an image:
  compute the median
  from all these medians, build a new image
Rectangular Array declaration and allocation

possible array declarations:
array [1:20, 1:30] of int Im;  // by first (lwb) and last (upb) valid indices
// first index (lwb) not necessarily 0 ,  why is this useful?
or
int [20,30] Im;                // lengths, first index, lwb, implied: 0
// last index (upb) length-1  C, JAVA

possible array allocations:
row major order  ( C, JAVA )
array starts with first row allocated consecutively
second row occurs directly after first, etc.
column major order  ( Fortran )
array starts with first column allocated consecutively
second column occurs directly after first, etc.
Array Example

```
array [1:20, 1:30] of int Im;

start: Im[1,1]  Im[1,30]
```

Assume row major order

\[ Im[2,1] \] directly follows \( Im[1,30] \)

Array elements contiguous region allocation:

\[
\text{start} = \text{malloc}(20 \times 30 \times \text{sizeof(int)})
\]

The descriptor should make element address calculation simple:

\[
&Im[i,j] = \text{start} + (i-\text{rowlwb}) \times \text{rowWidth} \\
&Im[i,j] = \text{base} + i \times \text{rowWidth} + j \times \text{colWidth}
\]

\[
base = ???
\]

**DESCRIPTOR IN GENERAL**

- \( dim_1: \text{lwb, upb, width} \)
- \( \ldots \)
- \( dim_n: \text{lwb, upb, width} \)

Start: \((30 \times 4 + 4)\)

\[
dim_1: 1, 20, 120 \\
dim_2: 1, 30, 4
\]

**DESCRIPTION: im**

\((\text{sizeof(int)} = 4)\)
array [1:20, 1:30] of int  \text{ Im;}

\begin{align*}
\text{Im}[1,1] & \quad \text{Im}[1,30] \\
\text{Im}[1,1] & \quad \text{Im}[2,1]
\end{align*}

Assume row major order:
\text{Im}[2,1] \text{ directly follows } \text{Im}[1,30]

Array elements contiguous region allocation:
\text{start} = \text{malloc}(20 \times 30 \times \text{sizeof(int)})

The descriptor should make element address calculation simple:
\&\text{Im}[i,j] = \text{base} + i \times \text{rowWidth} + j \times \text{colWidth}

\begin{align*}
\text{start-(30} & \times 4 + 4) \\
\text{dim}_1 & : 1, 20, 120 \\
\text{dim}_2 & : 1, 30, 4
\end{align*}

Given the descriptor for \text{Im}
Create descriptors for
\begin{enumerate}
\item array[1:20] of int column = \text{Im}[1:20,2]
\item array[1:2,1:2] of int window = \text{Im}[3:4,3:4]
\end{enumerate}
assuming start = 1000 \ (decimal)
Overview of full Meggy Java

Full MeggyJava Grammar
- Array declaration
- Array creation
- Array access and assignment
- Array length
- Meggy.setAuxLEDs

Game of Life Demo
- There is a MeggyJr grid off the resources page

PA6Rainbow example
class PA6rainbow { public static void main(String[] whatever){{ // display a rainbow on row 5  new Rainbow().run((byte)5); }}}

class Rainbow {
  Meggy.Color [] p;
  public void run(byte row) {
    p = new Meggy.Color [8];
    Meggy.setPixel((byte)2, (byte)3, p[0]);     Meggy.setPixel((byte)2, (byte)4, p[4]);
    this.displayRow(row, p);
  }
  public void displayRow(byte row, Meggy.Color [] a) {
    int i;  i=0;
    while (i<8) {
      Meggy.setPixel((byte)i, row, a[i]);        i = i+1;
    }
  }
}
## Implementing type checking for MeggyJava (Arrays)

**Syntax**

<table>
<thead>
<tr>
<th>Expression</th>
<th>AST Node(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>new int [ Exp ]</code></td>
<td><code>NewArrayExp</code></td>
</tr>
</tbody>
</table>
| `new Meggy.Color [Exp]` | `[LINENUM,POSNUM] Invalid operand type for new array operator`  
  // number of elements should be an integer or byte |
| `Exp [ Exp ]` | `ArrayExp and ArrayAssignStatement`  
  [LINENUM,POSNUM] Array reference to non-array type  
  [LINENUM,POSNUM] Invalid index expression type for array reference  
  // index expression should be of type integer or byte  
  [LINENUM,POSNUM] Invalid expression type assigned into array  
  // array could be an array of colors or an array of integers |
| `Exp . length` | `LengthExp`  
  [LINENUM,POSNUM] Operator length called on non-array type  
  // type of the length expression is integer |
Dynamically Allocating Arrays

**NewArrayExp**

- Assume size of array in elements is on stack as an int
- Gen code to calculate number of bytes, numelem * sizeof(elem)
- Gen code to add 2 bytes for length int to size of array
- Gen code to call malloc
- Gen code to set the first two bytes of array to numelem
- Gen code to push array’s address onto stack
Length Expression

outLengthExp

– Assume array reference/pointer is already on top of stack at runtime.
– Gen code that pops array reference off stack into two registers.
– Gen code that loads integer that array reference points to into two registers.
– Gen code to push that value/length onto the stack.
Array Elements  Array Assignment

outArrayExp

– Assume the integer index is at the top of the stack and the array reference/pointer is directly under it.
– Generate code to pop those off the stack and into registers.
– Generate code to calculate the array element address.
– Generate code that loads the array element and pushes it onto stack.

outArrayAssignStatement

– Assume that rhs expression, index expression, and array reference are on the stack.
– Generate code to pop those of the stack and store them into registers.
– Generate code that calculates the array element address (see previous slide).
– Generate code that stores the rhs expression into the array element memory location.