ARRAYS
Need for Arrays

• If I want to keep the score of 100 players in a game I could declare a separate variable to track each one’s score:
  – int player1 = N; int player2 = N; int player3 = N; ...
  – PAINFUL!!

• Enter arrays
  – Ordered collection of variables of the same type
  – Collection is referred to with **one name**
  – Individual elements referred to by an **offset/index** from the start of the array [in C, first element is at index 0]

• Example:
  – int player[100];
Arrays: Informal Overview

- Informal Definition:
  - Ordered collection of variables of the same type
- Collection is referred to with **one name**
- Individual elements referred to by an **offset/index** from the start of the array [in C, first element is at index 0]

```c
int data[20];
data[0] = 357;
data[1] = -1;
data[2] = 1035;
int x = data[0];
```

```c
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>420</td>
<td>424</td>
<td>428</td>
<td>496</td>
</tr>
<tr>
<td>357</td>
<td>-1</td>
<td>1035</td>
<td>…</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>420</td>
<td>421</td>
<td>422</td>
</tr>
<tr>
<td>‘h’</td>
<td>‘i’</td>
<td>00</td>
</tr>
</tbody>
</table>

Memory
Arrays – A First Look

• Formal Def: A **statically-sized, contiguously allocated collection of homogenous data elements**

• Collection of homogenous data elements
  – Multiple variables of the same data type

• Contiguously allocated in memory
  – One right after the next

• **Statically-sized**
  – Size of the collection can’t be changed after initial declaration/allocation

• Collection is referred to with **one name**

• Individual elements referred to by an **offset/index** from the start of the array [in C, first element is at index 0]
Example: Arrays

• Track amount of money (# of coins) 3 people have.
• Homogenous data set (number of coins) for multiple people...perfect for an array
  – int num_coins[3];
• Recall, memory has garbage values by default. You will need to initialized each element in the array

```c
int num_coins[3];
```

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
</tr>
<tr>
<td>204</td>
</tr>
<tr>
<td>208</td>
</tr>
<tr>
<td>212</td>
</tr>
<tr>
<td>216</td>
</tr>
<tr>
<td>220</td>
</tr>
<tr>
<td>224</td>
</tr>
<tr>
<td>228</td>
</tr>
<tr>
<td>232</td>
</tr>
<tr>
<td>236</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

num_coins[0]
num_coins[1]
num_coins[2]
Example: Arrays

- Track amount of money (# of coins) 3 people have.
- Homogenous data set (number of coins) for multiple people...perfect for an array
  - int num_coins[3];
- Must initialize elements of an array
  - for(int i=0; i < 3; i++)
    num_coins[i] = 0;

```
int num_coins[3];

200 00 00 00
204 00 00 00
208 00 00 00
212 AB AB AB AB
216 AB AB AB AB
220 AB AB AB AB
224 AB AB AB AB
228 AB AB AB AB
232 AB AB AB AB
236 AB AB AB AB
...
```

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>num_coins[0]</td>
</tr>
<tr>
<td>num_coins[1]</td>
</tr>
<tr>
<td>num_coins[2]</td>
</tr>
</tbody>
</table>
Arrays

• Track amount of money (# of coins) 3 people have.
• Homogenous data set (number of coins) for multiple people...perfect for an array
  – int num_coins[3];
• Must initialize elements of an array
  – for(int i=0; i < 3; i++)
    num_coins[i] = 0;
• Can access each persons amount and perform ops on that value
  – num_coins[0] = 5;
    num_coins[1] = 8;
Accessing elements

• While the size is fixed, index into array can be a variable (and usually is)

• If I have 100 players in my game I could declare a separate variable to track each one’s score:
  – int player1 = N; int player2 = N; int player3 = N; ...
  – PAINFUL!!

• Better idea: Use an array where the index to the desired element can be a variable:
  – int player[100];
  – for(i=0; i < 100; i++){
      player[i] = N;
  }

• Can still refer to individual items if desired: player[2]
ALGORITHMS
How Do You Find a Word in a Dictionary

• Describe an “efficient” method

• Assumptions / Guidelines
  – Let $target\_word = \text{word to lookup}$
  – $N$ pages in the dictionary
  – Each page has the $start$ and $last$ word on that page listed at the top of the page
  – Assume the user understands how to perform alphabetical (“lexicographic”) comparison (e.g. “abc” is smaller than “acb” or “abcd”)
Algorithms

• Algorithms are at the heart of computer systems, both in HW and SW
  – They are fundamental to Computer Science and Computer Engineering
• Informal definition
  – An algorithm is a precise way to accomplish a task or solve a problem
• Software programs are collections of algorithms to perform desired tasks
• Hardware components also implement algorithms from simple to complex
Humans and Computers

• Humans understand algorithms differently than computers
• Humans easily tolerate ambiguity and abstract concepts using context to help.
  – “Add a pinch of salt.” How much is a pinch?
  – “Michael Jordan could soar like an eagle.”
  – “It’s a bear market”
• Computers only execute well-defined instructions (no ambiguity) and operate on digital information which is definite and discrete (everything is exact and not “close to”)

Humans understand algorithms differently than computers because computers only execute well-defined instructions (no ambiguity) and operate on digital information which is definite and discrete (everything is exact and not “close to”). Humans, on the other hand, easily tolerate ambiguity and abstract concepts using context to help. For example:

- “Add a pinch of salt.” How much is a pinch?
- “Michael Jordan could soar like an eagle.”
- “It’s a bear market”
Formal Definition

- For a computer, “algorithm” is defined as...
  - ...an ordered set of unambiguous, executable steps that defines a terminating process

- Explanation:
  - **Ordered Steps**: the steps of an algorithm have a particular order, not just any order
  - **Unambiguous**: each step is completely clear as to what is to be done
  - **Executable**: Each step can actually be performed
  - **Terminating Process**: Algorithm will stop, eventually. (sometimes this requirement is relaxed)
Algorithm Representation

• An algorithm is not a program or programming language

• Just as a story may be represented as a book, movie, or spoken by a story-teller, an algorithm may be represented in many ways
  – Flow chart
  – Pseudocode (English-like syntax using primitives that most programming languages would have)
  – A specific program implementation in a given programming language
Pseudocode Primitives

• **Assignment:**
  
  \[ \text{name} \leftarrow \text{expression} \]
  
  – *name* is a descriptive name/variable and *expression* describes the value to be associated with *name*

• **Select one of two possible choices (conditionals):**
  
  \[
  \begin{align*}
  \text{if (condition) then (activity) else (activity)} \\
  \text{if (condition) then (activity)}
  \end{align*}
  \]

• **Repeated execution of statements (loops):**
  
  \[
  \begin{align*}
  \text{while (condition) do (activity)} \\
  \text{repeat (activity) until (condition)} \\
  \text{foreach name in (set / collection) do (activity)}
  \end{align*}
  \]
Syntax and Semantics

• **Syntax**: refers to the primitive’s symbolic representation – the “proper” way to write the primitive

• **Semantics**: the “meaning” of the primitive

• Example: ‘air’
  – The syntax consists of the 3 symbols ‘a’, ‘i’, and ‘r’. The semantics of this word is “a gaseous substance that surrounds the earth”

• Code Example
  – **if** (condition) **then** (activity) is the syntax.
  – The semantics (meaning) is “the *activity* will only be performed if *condition* is true”
Indentation

• Pseudocode (and programs in real programming languages) are usually indented to enhance readability by humans

1. if (not raining) then (if (temperature == hot) then (go swimming) else (play golf)) else (watch television)

2. if (not raining) then
   (if (temperature == hot) then
    (go swimming)
    else (play golf))
else (watch television)

Which is easier to read?
Algorithm Example 1

• List/print all factors of a natural number, \( n \)
  – How would you check if a number is a factor of \( n \)?
  – What is the range of possible factors?
  \[
i \leftarrow 1
\]
  \[
\text{while}(i \leq n) \; \text{do}
\]
  \[
\quad \text{if} \; (\text{remainder of } n/i \; \text{is zero}) \; \text{then}
\]
  \[
\quad \quad \text{List } i \; \text{as a factor of } n
\]
  \[
\quad i \leftarrow i+1
\]

• An improvement
  \[
i \leftarrow 1
\]
  \[
\text{while}(i \leq \sqrt{n}) \; \text{do}
\]
  \[
\quad \text{if} \; (\text{remainder of } n/i \; \text{is zero}) \; \text{then}
\]
  \[
\quad \quad \text{List } i \; \text{and } n/i \; \text{as a factor of } n
\]
  \[
\quad i \leftarrow i+1
\]
Algorithm Time Complexity

• We often judge algorithms by how long they take to run for a given input size.
• Algorithms often have different run-times based on the input size [e.g. # of elements in a list to search or sort]
  – Different input patterns can lead to best and worst case times.
  – Average-case times can be helpful, but we usually use worst case times for comparison purposes.
Big-O Notation

- Given an input to an algorithm of size $n$, we can derive an expression in terms of $n$ for its worst case run time (i.e. the number of steps it must perform to complete)
- From the expression we look for the dominant term and say that is the big-O (worst-case or upper-bound) run-time
  - If an algorithm with input size of $n$ runs in $n^2 + 10n + 1000$ steps, we say that it runs in $O(n^2)$ because if $n$ is large $n^2$ will dominate the other terms

```plaintext
i ← 1
while(i <= n) do
  if (remainder of n/i is zero) then
    List i as a factor of n
  i ← i+1
```

$1 = O(n)$
$1\cdot n$
$2\cdot n$
$1\cdot n$
$1\cdot n$

$5n+1 = O(n)$
Big-O Notation

- Given an input to an algorithm of size $n$, we can derive an expression in terms of $n$ for its worst case run time (i.e. the number of steps it must perform to complete)
- From the expression we look for the dominant term and say that is the big-O (worst-case or upper-bound) run-time
  - If an algorithm with input size of $n$ runs in $n^2 + 10n + 1000$ steps, we say that it runs in $O(n^2)$ because if $n$ is large $n^2$ will dominate the other terms
- Main sources of run-time: Loops
  - Even worse: Loops within loops (i.e. execute all of loop 2 w/in a single iteration of loop 1, and repeat for all iterations of loop 1, etc.)

```
i ← 1
while(i <= n) do
  if (remainder of n/i is zero) then
    List i as a factor of n
  i ← i+1
```

1  
1*n  
2*n  
1*n  
1*n  
5n+1  
= O(n)
Algorithm Example 1

- List/print all factors of a natural number, $n$
  - What is a factor?
  - What is the range of possible factors?
  
  \[
i \leftarrow 1
  \]
  
  \[
  \text{while}(i \leq n) \text{ do}
  \]
  
  \[
  \quad \text{if (remainder of } n/i \text{ is zero) then}
  \]
  
  \[
  \quad \text{List } i \text{ as a factor of } n
  \]
  
  \[
  \quad i \leftarrow i+1
  \]
  
  \[
  O(n)
  \]

- An improvement
  
  \[
i \leftarrow 1
  \]
  
  \[
  \text{while}(i \leq \sqrt{n}) \text{ do}
  \]
  
  \[
  \quad \text{if (remainder of } n/i \text{ is zero) then}
  \]
  
  \[
  \quad \text{List } i \text{ and } n/i \text{ as a factor of } n
  \]
  
  \[
  \quad i \leftarrow i+1
  \]
  
  \[
  O(\sqrt{n})
  \]
Algorithm Example 2a

- Searching an ordered list (array) for a specific value, k, and return its index or -1 if it is not in the list

- Sequential Search
  - Start at first item, check if it is equal to k, repeat for second, third, fourth item, etc.

```plaintext
i ← 0
while ( i < length(myList) ) do
  if (myList[i] equal to k) then return i
  else i ← i+1
return -1
```
Algorithm Example 2b

• Sequential search does not take advantage of the ordered nature of the list
  – Would work the same (equally well) on an ordered or unordered list

• Binary Search
  – Take advantage of ordered list by comparing k with middle element and based on the result, rule out all numbers greater or smaller, repeat with middle element of remaining list, etc.

```
List: 2 3 4 6 9 10 13 15 19
Index: 0 1 2 3 4 5 6 7 8

Start in middle

k = 6

6 < 9

List: 2 3 4 6 9 10 13 15 19
Index: 0 1 2 3 4 5 6 7 8

6 > 4

List: 2 3 4 6 9 10 13 15 19
Index: 0 1 2 3 4 5 6 7 8

6 = 6
```
Algorithm Example 2b

**Binary Search**
- Compare \( k \) with middle element of list and if not equal, rule out \( \frac{1}{2} \) of the list and repeat on the other half
- Implementation:
  - Define range of searchable elements = \([\text{start}, \text{end})\)
  - (i.e. start is inclusive, end is exclusive)

\[
\text{start} \leftarrow 0; \text{end} \leftarrow \text{length(List)}; \\
\textbf{while} (\text{start index not equal to end index}) \textbf{do} \\
\hspace{1em} i \leftarrow (\text{start} + \text{end}) / 2; \\
\hspace{1em} \textbf{if} ( k = \text{List}[i] ) \textbf{then return} i; \\
\hspace{1em} \textbf{else if} ( k > \text{List}[i] ) \textbf{then start} \leftarrow i+1; \\
\hspace{1em} \textbf{else end} \leftarrow i; \\
\hspace{1em} \text{return} -1;
\]
Sorting

• If we have an unordered list, sequential search becomes our only choice
• If we will perform a lot of searches it may be beneficial to sort the list, then use binary search
• Many sorting algorithms of differing complexity (i.e. faster or slower)
• Bubble Sort (simple though not terribly efficient)
  – On each pass through thru the list, pick up the maximum element and place it at the end of the list. Then repeat using a list of size n-1 (i.e. w/o the newly placed maximum value)
Bubble Sort Algorithm

\[ n \leftarrow \text{length(List)}; \]
\[ \text{foreach } i \text{ in } n-2 \text{ downto } 1 \text{ do} \]
\[ \text{foreach } j \text{ in } 0 \text{ to } i \text{ do} \]
\[ \text{if } (\text{List}[j] > \text{List}[j+1]) \text{ then} \]
\[ \text{swap List}[j] \text{ and List}[j+1] \]

Pass 1

\[
\begin{array}{cccccc}
7 & 3 & 8 & 6 & 5 & 1 \\
\text{j} & \text{i} \\
3 & 7 & 8 & 6 & 5 & 1 \\
\text{j} & \text{i} \\
3 & 7 & 8 & 6 & 5 & 1 \\
\text{j} & \text{i} \\
3 & 7 & 6 & 8 & 5 & 1 \\
\text{j} & \text{i} \\
3 & 7 & 6 & 5 & 8 & 1 \\
\text{i},\text{j} \\
3 & 7 & 6 & 5 & 1 & 8 \\
\text{swap} \\
\end{array}
\]

Pass 2

\[
\begin{array}{cccccc}
3 & 7 & 6 & 5 & 1 & 8 \\
\text{j} & \text{i} \\
3 & 7 & 6 & 5 & 1 & 8 \\
\text{j} & \text{i} \\
3 & 6 & 7 & 5 & 1 & 8 \\
\text{i},\text{j} \\
3 & 6 & 5 & 7 & 1 & 8 \\
\text{swap} \\
3 & 6 & 5 & 1 & 7 & 8 \\
\text{swap} \\
\end{array}
\]

Pass n-1

\[
\begin{array}{cccccc}
1 & 3 & 5 & 6 & 7 & 8 \\
\text{i} \\
1 & 3 & 5 & 6 & 7 & 8 \\
\text{i},\text{j} \\
1 & 3 & 5 & 6 & 7 & 8 \\
\text{swap} \\
\end{array}
\]
Complexity of Search Algorithms

- **Sequential Search**: List of length n
  - Worst case: Search through entire list
  - Time complexity = an + k
    - a is some constant for number of operations we perform in the loop as we iterate
    - k is some constant representing startup/finish work (outside the loop)
  - Sequential Search = \( O(n) \)

- **Binary Search**: List of length n
  - Worst case: Continually divide list in two until we reach sublist of size 1
  - Time = a*\( \log_2 n \) + k = \( O(\log_2 n) \)

- As \( n \) gets large, binary search is far more efficient than sequential search
Complexity of Sort Algorithms

• Bubble Sort
  – 2 Nested Loops
  – Execute outer loop n-1 times
  – For each outer loop iteration, inner loop runs i times.
  – Time complexity is proportional to:
    \[ n-1 + n-2 + n-3 + \ldots + 1 = (n^2 + n)/2 = O(n^2) \]
• Other sort algorithms can run in \( O(n \times \log_2 n) \)
Importance of Time Complexity

- It makes the difference between effective and impossible
- Many important problems currently can only be solved with exponential run-time algorithms (e.g. $O(2^n)$ time)...we call these NP = Non-deterministic polynomial time algorithms) [No known polynomial-time algorithm exists]
- Usually algorithms are only practical if they run in $P = \text{polynomial time}$ (e.g. $O(n)$ or $O(n^2)$ etc.)
- One of the most pressing open problems in CS: “Is NP = P?”
  - Do P algorithms exist for the problems that we currently only have an NP solution for?

<table>
<thead>
<tr>
<th>N</th>
<th>O(1)</th>
<th>O(log$_2$n)</th>
<th>O(n)</th>
<th>O(n*log$_2$n)</th>
<th>O(n$^2$)</th>
<th>O(2$^n$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>4.3</td>
<td>20</td>
<td>86.4</td>
<td>400</td>
<td>1,048,576</td>
</tr>
<tr>
<td>200</td>
<td>1</td>
<td>7.6</td>
<td>200</td>
<td>1,528.8</td>
<td>40,000</td>
<td>1.60694E+60</td>
</tr>
<tr>
<td>2000</td>
<td>1</td>
<td>11.0</td>
<td>2000</td>
<td>21,931.6</td>
<td>4,000,000</td>
<td>#NUM!</td>
</tr>
</tbody>
</table>