CST 280 Advanced C++ Programming
Week 2

Topics:

- More on C++ functions
- C++ random numbers
- Searching and sorting lists
- List algorithms

Topic:
More on C++ Functions

- Returning a Boolean value
- Overloading functions
- Function stubs and drivers
- Preconditions and postconditions

Boolean Data Type

- A data type that only allows storage of a 'True' or 'False'
- Often used as a "switch" or "flag"
- Example:

```
bool done;
done = false;

// do work
done = true;

if (done) ...
```

File: `testEven.cpp`

Function *Overloading*

- Like-named functions that can be defined as different actions based on:
  - data type of parameters
  - number of parameters

Function Overloading: Example

```
// Summing 2 values
int sum(int num1, int num2)

// Summing 3 values
int sum(int num1, int num2, int num3)

// Summing 4 values
int sum(int num1, int num2, int num3, int num4)
```

Function Overloading: Example

```
void swap (double& x, double& y);

void swap (int& x, int& y);

void swap (char& x, char& y);
```

•
•
•

Question

- **How would you implement:**

```
void swap (int& x, int& y)
{




}
```

Program Example

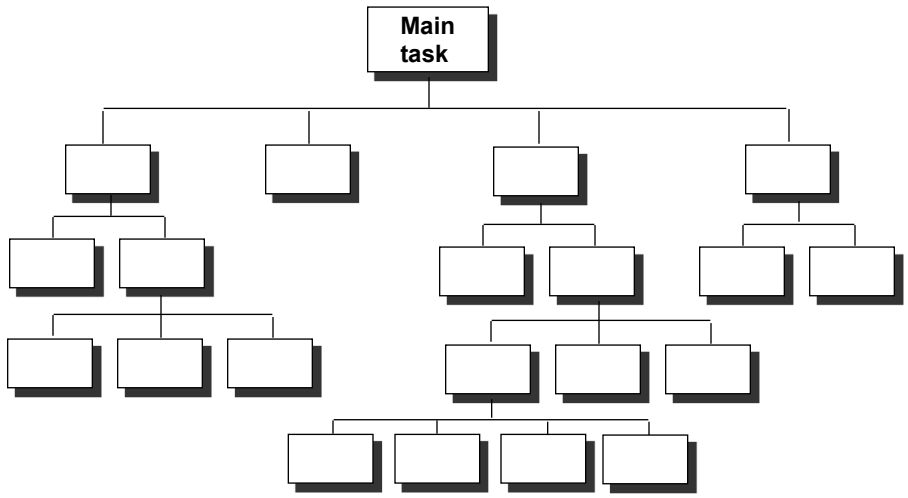
- **Function overloading:**
  - **Functions:**

    sum

    swap

File: **funOverload.cpp**

Top-Down Function Design Implementation Strategies



Approaches to Function Implementation

TOP-DOWN	BOTTOM-UP
Ensures correct overall design logic.	**Ensures individual modules work together correctly, beginning with the lowest level.**
USES: placeholder module *stubs* to test the order of calls.	**USES: a test *driver* used to call the functions being tested.**

Value of Drivers & Stubs

- Allows programmer to "get something running" without completion of entire program
- Enables an *incremental approach* to code development
- Very typical to use with good top-down design

Function Design and Documentation: Preconditions and Postconditions

- The precondition is an assertion describing what a function requires to be true before beginning execution.
- The postcondition describes what must be true at the moment the function finishes execution.
- The *caller* is responsible for ensuring the precondition, and the *function code* must ensure the postcondition.

FOR EXAMPLE . . .

```
//*********************************************************
// The binarySearch function performs a binary search on an
// integer array. The array, which has a maximum of numelems
// elements, is searched for the number stored in value. If
// the number is found, its array subscript is returned.
// Otherwise, -1 is returned indicating the value was not in
//  the array.
//  Pre:   list in array has been initialized and is ORDERED
//         number of elements is correct
//  Post:  list in array is unchanged
//          variable value has either -1 (for not found) or
//                   value 0 … numelems-1
int binarySearch(int array[], int numelems, int value)
{
       •
       •
       •
```

Preconditions and Postconditions

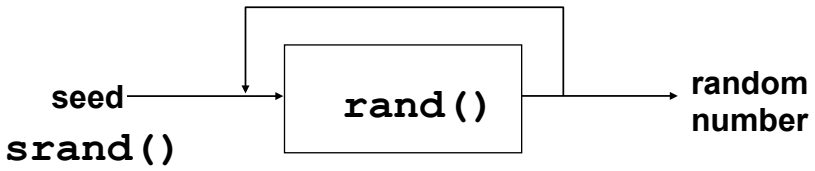
- Very useful for documenting function interfaces
- Does not define how a function does its work
- A "contract" between function author and user of function

Topic:
C++ Random Numbers

- **Definition and use**

Random Number Functions

- **A value selected by chance**
- **Generated by a mathematical function implemented in C++**
- **Behavior:**



File: **`randDemo.cpp`**

Applications of Random Numbers

- **Useful for simulations, experiments, and demonstrations**
- **Necessary for random behavior required for game events**

How Do You Simulate a Die Throw?

Program Demos

- Using random numbers:
  - Dice simulation
  - State capital game

Two Dice Simulation

- Simulate frequency distribution from throwing two dice
- Use arrays as containers for counters

File: `diceSim.cpp`

State Capitals Game

- Select index random list of state names
- Test user input against parallel array of state capital names

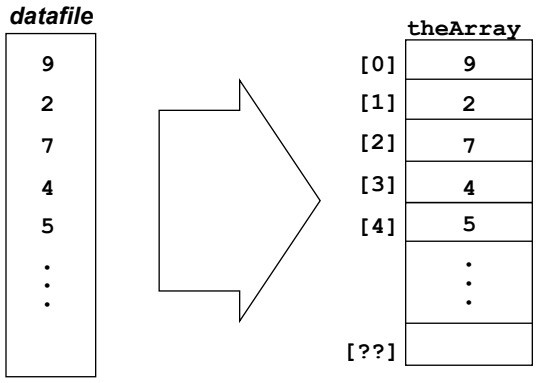
File: `stateCapGame.cpp`

Topic:
List Algorithms using Arrays

- Review: arrays from files; parallel arrays
- List processing concepts
- Searching algorithms
- List addition/deletion algorithms

Review: Array Input to File

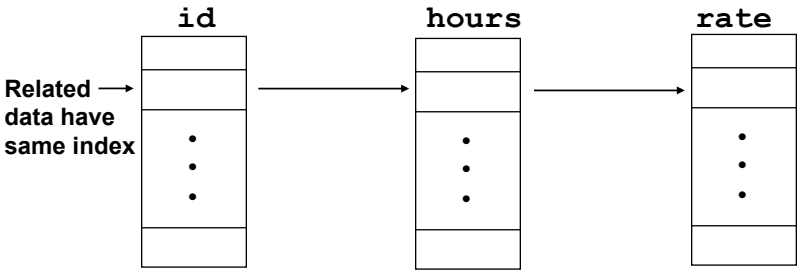
- **Read an unknown number of integer values from a file into an array**
- **Read the array and calculate the average of the list of values**



File: `aveFileArray.cpp`

Parallel Arrays

- **Multiple arrays; same size; related elements of different data types**
- **Example:**



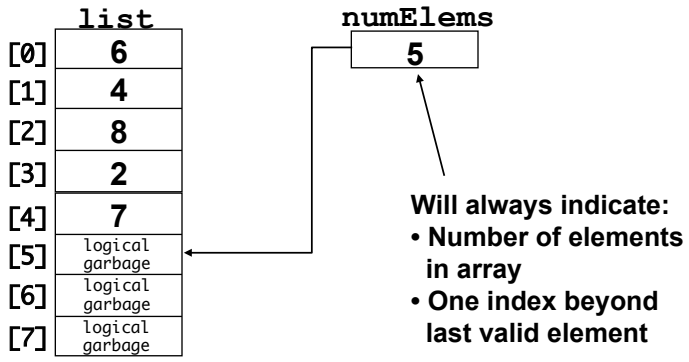
Program Demo: `parallel.cpp`

List Processing

- A simple list is a common data structure often utilized in applications
- Arrays often used for list storage
- Lists may be:
  - Unordered
  - Ordered
- Algorithms require management of:
  - Size of data list
  - Maximum possible list size (array dimension)
- Allows direct access to all list elements (using index)

Model of a List Data Structure

```
int list[8];
```



Operations on Lists

- Search
- Sort
- Add to
- Delete from
- Change values in
- Perform calculations on
- Read from file
- Write to file
- Print

Searching Algorithms

- Finding information in a list *efficiently*
- Involve various strategies for traversing a list of value to match a search target
- Often searching for a *key* value
- Useful for database applications

Linear Search

- **What is the strategy?**

Linear Search

```
int searchList(int list[], int numElems, int value)
{
    int index = 0;          // Used as a subscript to search array
    int position = -1;      // To record position of search value
    bool found = false;     // Flag to indicate if the value was found

    while (index < numElems && !found)
    {
        if (list[index] == value)    // If the value is found
        {
            found = true;            // Set the flag
            position = index;        // Record the value's subscript
        }
        index++;                     // Go to the next element
    }
    return position;                 // Return the position, or -1
}
```

Returns either:

- **Index of element searched for**
- **A special index code for a failed search (`-1`)**

Demo: **`linSearch.cpp`**

Example: Searching for Information in Parallel Arrays

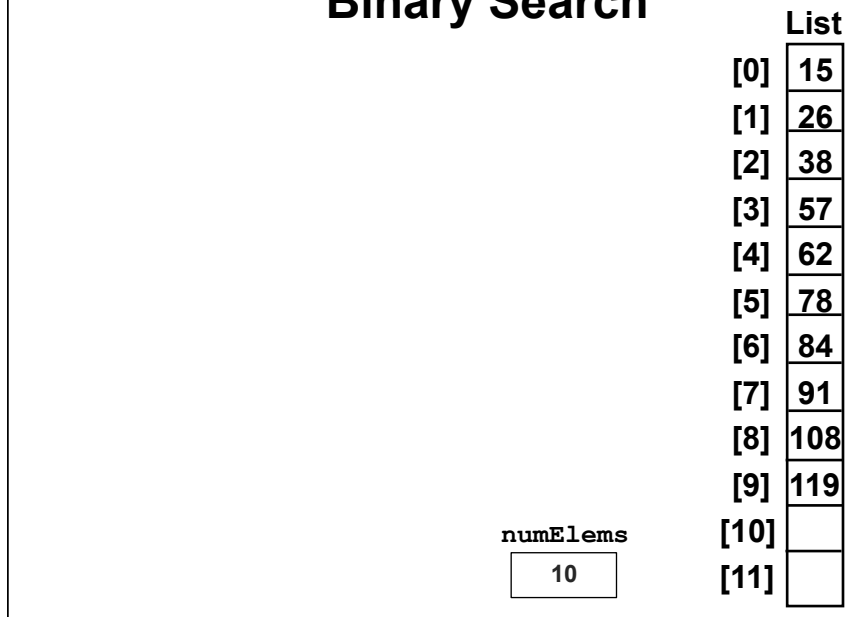
- A simple student database – parallel lists with:
  - `studentID` `grade` `GPA`
- Action:
  - Prompt user for a student ID
  - Search to find index of matching student ID
  - Write grade and GPA of student
- Note: use of modular programming:
  - Linear search "module" is "plugged in" to this new and different applications with no changes

Program Demo - file: `parallelSearch.cpp`

Binary Search

- A much more efficient method for searching
- Do you recall:
  - Strategy?
  - Precondition?

Binary Search



Binary Search: Manual Trace

theArray

12	26	34	57	61	76	84	91	102	130
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
First				Middle					Last

Search for: 1) 102

2) 29

```
int binarySearch(int array[], int numelems, int value)
{
    int first = 0,              // First array element
        last = numelems - 1,    // Last array element
        middle,                 // Mid point of search
        position = -1;          // Position of search value
    bool found = false;         // Flag

    while (!found && first <= last)
    {
        middle = (first + last) / 2;    // Calculate mid point
        if (array[middle] == value)     // If value is found at mid
        {
            found = true;
            position = middle;
        }
        else if (array[middle] > value)  // If value is in lower half
            last = middle - 1;
        else
            first = middle + 1;          // If value is in upper half
    }
    return position;
}
```

File: **binSearch.cpp**

Searching Algorithms: Efficiency

- **Example of efficiency gained: search 1024 elements**
  - **Linear search: worst case = _____ compares**
  - **Binary search: worst case = _____ compares**
- **But, binary search requires ordered list which entails additional list management**

Experiment: Efficiency of Searching Algorithms

- Define array containing integers 1,2,3,…,99,100
- Select random number 1…100 to search for
- Measure number of comparisons required to find target
- Perform 1000 searches and average total

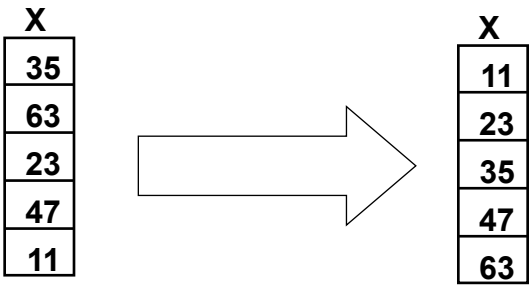
Files: `binSearchMeasure.cpp`

Topic: Basic Sorting Algorithms

- Bubble Sort
- … others to come …

Sorting means . . .

- The values stored in an array have *keys* of a type for which the relational operators are defined (assume *unique* keys)
- Sorting rearranges the elements into either *ascending* or *descending* order within the array. (below is ascending order)



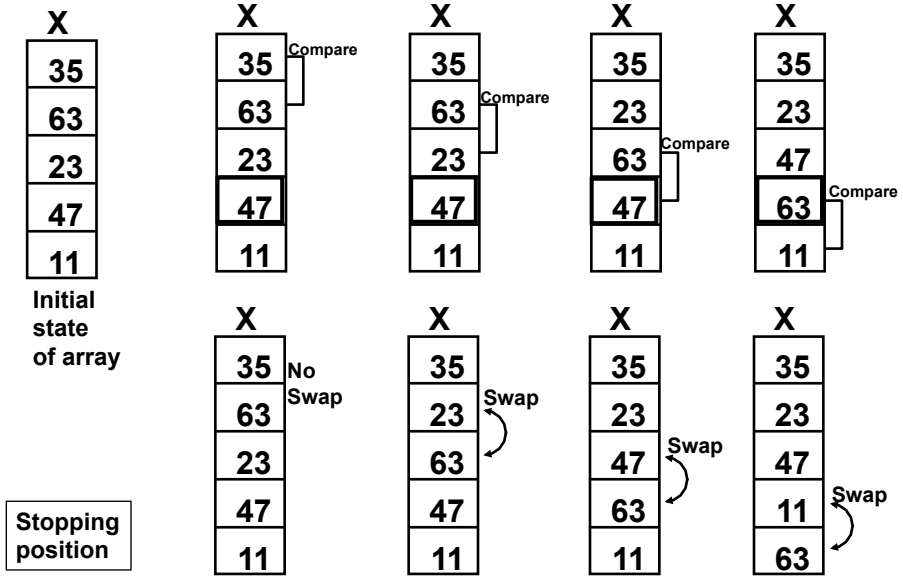
Sorting Algorithms

- Ordering an unordered list
- Order can be:
  - Ascending: small-to-large
  - Descending: large-to-small
- Basic sorting algorithms:
  - Bubble sort
  - … and several more covered late …
- Algorithm strategies often include a pattern of array element *swaps*

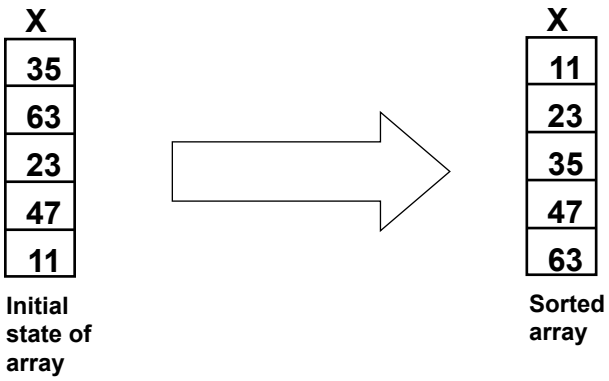
Bubble Sort

- A very basic sorting algorithm
- General strategy:
  - Examine *adjacent pairs* of values and swap if out of order
  - Elements swapped in such a way that larger elements "bubble up" to the bottom of the array

Bubble Sort: Pass 1



Bubble Sort: Upon completion



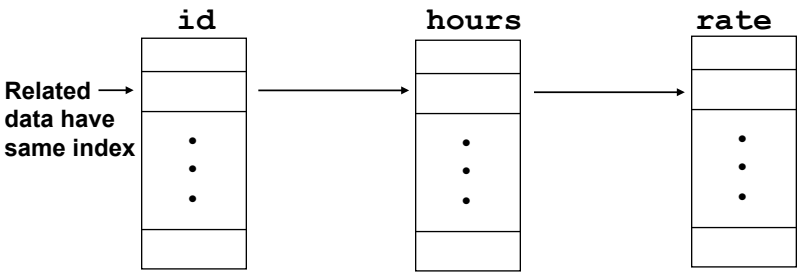
Bubble Sort

```
void sortArray(int array[], int elems)
{
   int temp, end;

   for (end = elems - 1; end >= 0; end--)
   {
      for (int count = 0; count < end; count++)
      {
         if (array[count] > array[count + 1])
         {
            temp = array[count];
            array[count] = array[count + 1];
            array[count + 1] = temp;
         }
      }
   }
}
```

File: **bubble.cpp**

Sorting Parallel Arrays

- Sorting parallel arrays requires careful swaps of array elements
- Example:



Program Demo: `parallelSort.cpp`

Topic: List Processing Algorithms

- Generally adding/removing items from ordered/unordered lists
- Related to database actions
- Additional, more specialized algorithms presented

Files and Arrays

Algorithm to build an array from a file:

```
void GetList(int inList[],int& listsize,int maxlistsize)
{
    ifstream InFile ("fileName.txt");
    int ListElement;

    int i = 0;
    InFile >> ListElement;
    while( ! InFile.eof() && i < maxlistsize)
    {
        inList[i] = ListElement;
        i++;
        InFile >> ListElement;
    }
    listsize = i;        // Size of list is lastindex + 1
}
```

Additional List Algorithms

- **Concepts of a "list" can be further formalized with additional algorithms, mainly to:**
  - **Add to a list**
  - **Delete from a list**

Write algorithms for ...

… inserting into an unordered list

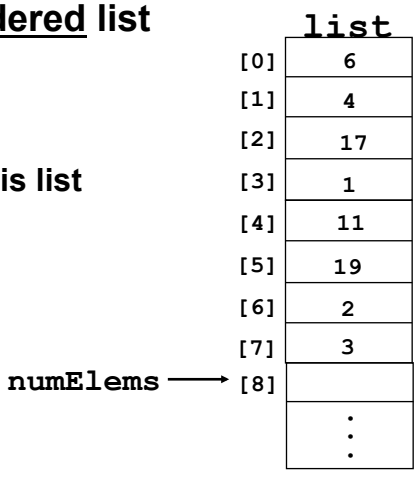
… deleting from an unordered list

Write algorithm for ...

… inserting into an unordered list

Insert the value 10 into this list

newint 10

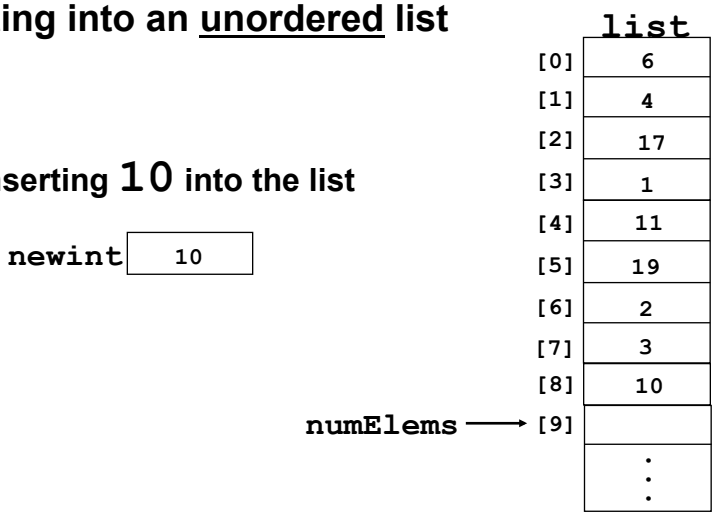


Inserting into an Unordered List

Verification ...

… inserting into an unordered list

After inserting 10 into the list

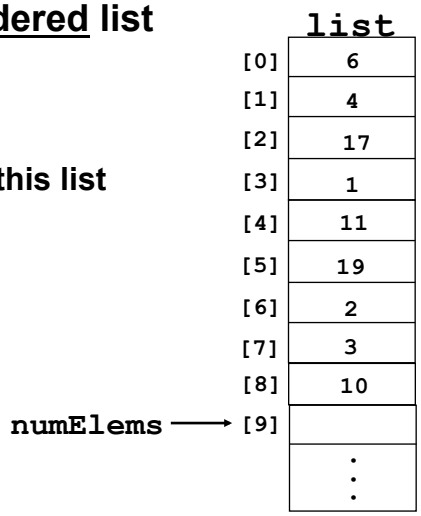


Write algorithm for ...

… deleting from an unordered list

Delete the value 19 from this list

oldint 19

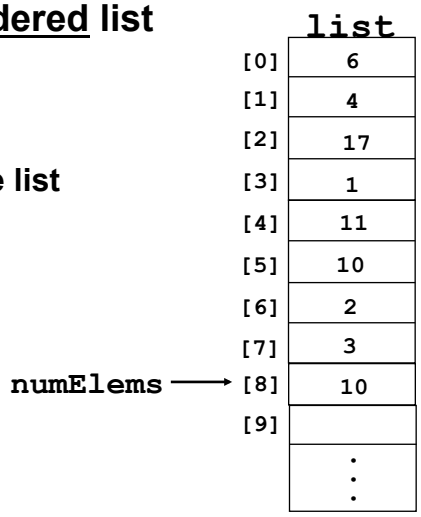


Verification ...

… deleting from an unordered list

After deleting 19 from the list

oldint 19



Program Demo

- **Test driver for inserting and deleting items using an unordered list**
- **File: `unorder.cpp`**

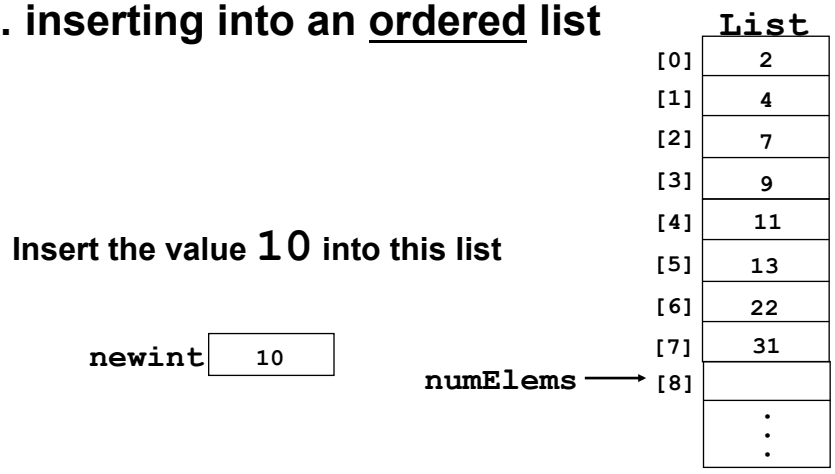
Practice:

- **Develop algorithm for to the front of an unordered list**

```
void UnOrdInsertFront(int list[], int& numElems, int newint)
{









}
```

Write algorithm for ...

… inserting into an ordered list



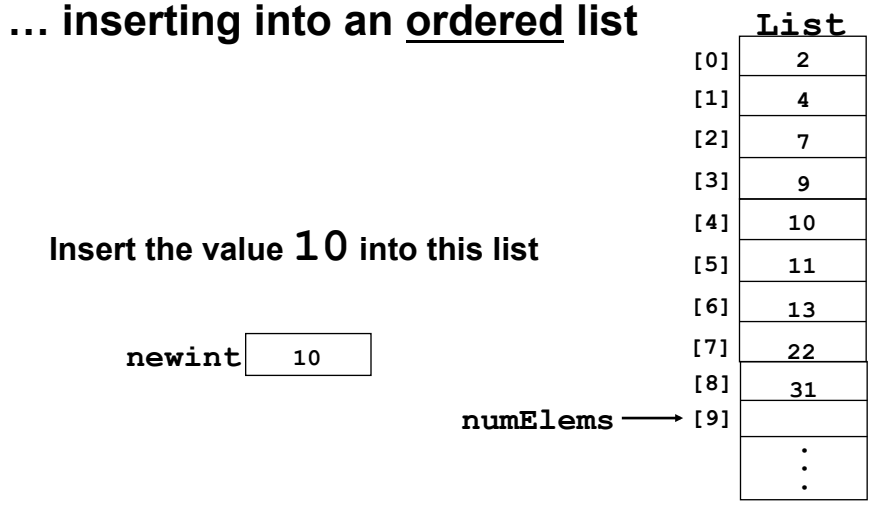
Practice:

- **Develop algorithm for inserting into an ordered list**

```
void OrdListInsert(int list[], int& numElems, int newint)
{









}
```

Verification for ...

… inserting into an ordered list



Deleting from an Ordered List

Program Demo

- Test driver for inserting and deleting items using an unordered list
- File: `order.cpp`

Experiment

- Goal: maintain a sorted list efficiently
- Measure differences between:
  1. Inserting a value and sort
  2. Inserting into ordered list position
  3. Inserting all values in an unordered list and then sort
- Note: What about efficiency of sorting algorithm?

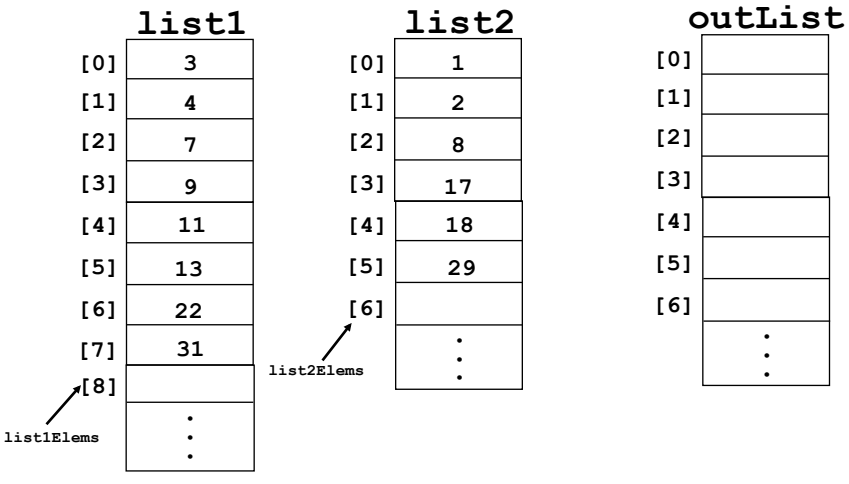
Files: **`keepSortMeasure.cpp`**

Other List Algorithms

- Merging two ordered lists
- Reversing a list
- List processing with parallel arrays

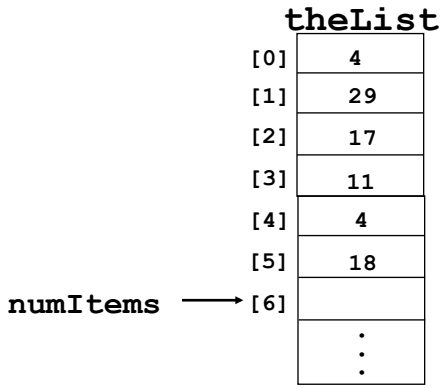
Algorithm Development: Merging Two Ordered Lists

- Develop algorithm (precondition: no duplicates)



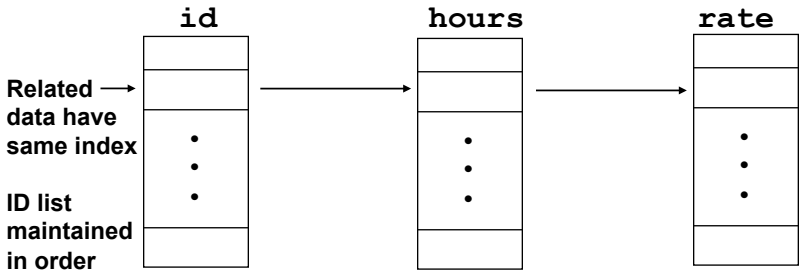
Algorithm Development: Reversing an Unordered List

- **Develop algorithm**



Example: Inserting into An Ordered List (with Parallel Arrays)

- **Multiple arrays; same size; related elements of different data types**
- **Example:**



File: `parallelInsert.cpp`

Analysis

- **What has to change if data type of list changes from `int` to something else?**
- **Can we create generic list algorithms (and later a *class*)?**

Files:

`ItemType.h`

`ListType.h`

`ListType.cpp`

`orderCharList.cpp`