Arrays

Mechanism for representing lists

Lists

- Problem solving often requires information be viewed as a list
  - List may be one-dimensional or multidimensional
- C++ provides two list mechanisms
  - Arrays
    - Traditional and important because of legacy libraries
    - Restrictions on its use
  - Container classes
    - First-class list representation
    - Common containers provided by STL
      - Vector, queue, stack, map, …
    - Preferred long-term programming practice

Array Terminology

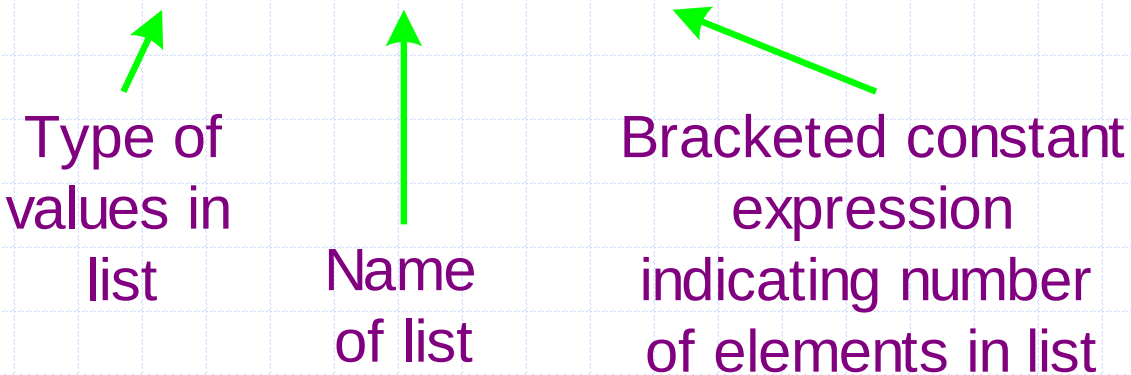
- List is composed of *elements*
- Elements in a list have a *common name*
  - The list as a whole is referenced through the common name
- List elements are of the same type — the *base* type
- Elements of a list are referenced by *subscripting* or *indexing* the common name

C++ Restrictions

- Subscripts are denoted as expressions within brackets: [ ]
- Base type can be any fundamental, library-defined, or programmer--defined type
- The index type is integer and the index range must be 0 ... `n`-1
  - where `n` is a programmer-defined constant expression.
- Parameter passing style
  - Always call by reference (no indication necessary)

Basic Array Definition

```
BaseType Id [ SizeExp ] ;
```




```
double X [ 100 ] ;

                    // Subscripts are 0 through 99
```

Example Definitions

- Suppose

```
const int N = 20;
const int M = 40;
const int MaxStringSize = 80;
const int MaxListSize = 1000;
```

- Then the following are all correct array definitions

```
int A[10];                    // array of 10 ints
char B[MaxStringSize];        // array of 80 chars
double C[M*N];                // array of 800 floats
int Values[MaxListSize];      // array of 1000 ints
Rational D[N-15];             // array of 5 Rationals
```

Subscripting

- Suppose

```
int A[10];     // array of 10 ints A[0], … A[9]
```

- To access individual element must apply a subscript to list name **A**
  - A subscript is a bracketed expression also known as the index
  - First element of list has index 0

```
A[0]
```

  - Second element of list has index 1, and so on

```
A[1]
```

  - Last element has an index one less than the size of the list

```
A[9]
```

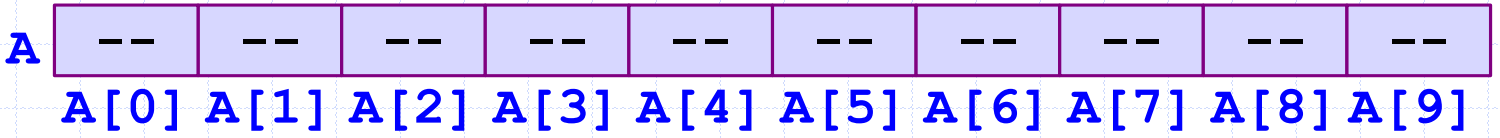
  - Incorrect indexing is a common error

```
A[10]      // does not exist
```

Array Elements

- Suppose

```
int A[10];    // array of 10 uninitialized ints
```

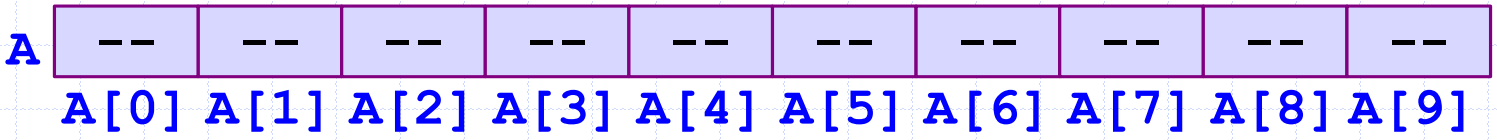


- To access an individual element we must apply a subscript to list name `A`

Array Element Manipulation

- Consider

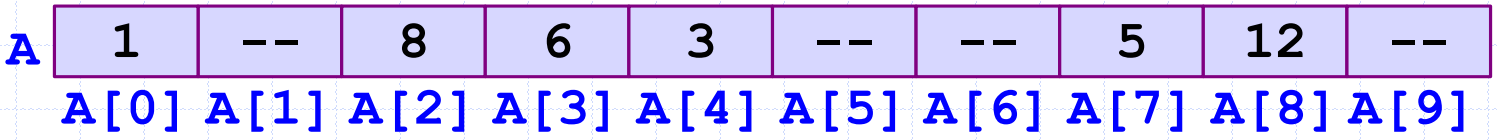
```
int i = 7, j = 2, k = 4;
A[0] = 1;
A[i] = 5;
A[j] = A[i] + 3;
A[j+1] = A[i] + A[0];
A[A[j]] = 12;
cin >> A[k]; // where next input value is 3
```



Array Element Manipulation

- Consider

```
int i = 7, j = 2, k = 4;
A[0] = 1;
A[i] = 5;
A[j] = A[i] + 3;
A[j+1] = A[i] + A[0];
A[A[j]] = 12;
cin >> A[k]; // where next input value is 3
```



Extracting Values For A List

```
int A[MaxListSize];
int n = 0;
int CurrentInput;
while((n < MaxListSize) && (cin >> CurrentInput)){
   A[n] = CurrentInput;
   ++n;
}
```

Displaying A List

```
// List A of n elements has already been set
for (int i = 0; i < n; ++i) {
   cout << A[i] << " ";
}
cout << endl;
```

Smallest Value

- Problem
  - Find the smallest value in a list of integers
- Input
  - A list of integers and a value indicating the number of integers
- Output
  - Smallest value in the list
- Note
  - List remains unchanged after finding the smallest value!

Preliminary Design

- Realizations
  - When looking for value with distinguishing characteristics, need a way of remembering best candidate found so far
    - Make it a function -- likely to be used often
- Design
  - Search array looking for smallest value
    - Use a loop to consider each element in turn
    - If current element is smallest so far, then update smallest value so far candidate
  - When done examining all of the elements, the smallest value seen so far is the smallest value

Passing An Array

Notice brackets are empty

```
int ListMinimum(const int A[], int asize) {
   assert(asize >= 1);
   int SmallestValueSoFar = A[0];
   for (int i = 1; i < asize; ++i) {
      if (A[i] < SmallestValueSoFar ) {
         SmallestValueSoFar = A[i];
      }
   }
   return SmallestValueSoFar ;
}
```

Could we just assign a 0 and have it work?

Using ListMinimum()

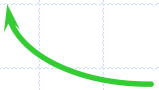
- What happens with the following?

```
int Number[6];
Number[0] = 3; Number[1] = 88; Number[2] = -7;
Number[3] = 9; Number[4] = 1;  Number[5] = 24;

cout << ListMinimum(Number, 6) << endl;

int List[3];
List[0] = 9;   List[1] = 12;   List[2] = 45;

cout << ListMinimum(List, 3) << endl;
```



Notice no brackets

Remember

- Arrays are always passed by reference
  - Artifact of C
- Can use `const` if array elements are not to be modified
- Do not need to include the array size when defining an array parameter

Some Useful Functions

```
void DisplayList(const int A[], int n) {
    for (int i = 0; i < n; ++i) {
        cout << A[i] << " ";
    }
    cout << endl;
}
void GetList(int A[], int &n, int MaxN = 100) {
   for (n = 0; (n < MaxN) && (cin >> A[n]); ++n) {
        continue;
    }
}
```

Useful Functions Being Used

```
const int MaxNumberValues = 25;
int Values[MaxNumberValues];
int NumberValues;

GetList(Values, NumberValues, MaxNumberValues);
DisplayList(Values, NumberValues);
```

Searching

- Problem
  - Determine whether a value key is one of the element values
- Does it matter if
  - Element values are not necessarily numbers
  - Element values are not necessarily unique
  - Elements may have key values and other fields

Sequential List Searching

```
int Search(const int List[], int m, int Key) {
    for (int i = 0; i < m; ++i) {
        if (List[i] == Key) {
            return i;
        }
    }
    return m;
}
```

Run time is proportional to number of elements

Example Invocation

```
cin >> val;
int spot = Search(Values, NumberValues, val);
if (spot != NumberValues) {
   // its there, so display it
   cout << Values[spot] << endl;
}
else { // its not there, so add it
   Values[NumberValues] = val;
   ++NumberValues;
}
```

Sorting

- Problem
  - Arranging elements so that they are ordered according to some desired scheme
    - Standard is non-decreasing order
      - Why don't we say increasing order?
- Major tasks
  - Comparisons of elements
  - Updates or element movement

Common Sorting Techniques

- Selection sort
  - In the i-th iteration, place the i-th smallest element in the i-th list location
- Bubble sort
  - Iteratively pass through the list examining adjacent pairs of elements and if necessary swap them to put them in order. Repeat the process until no swaps are necessary

Common Sorting Techniques

- Insertion sort
  - In i-th iteration, place the i-th element with respect to the i-1 previous elements
- Quick sort
  - Divide the list into sublists such that every element in the left sublist <= to every element in the right sublist. Repeat the Quick sort process on the sublists

SelectionSort

```
void SelectionSort(int A[], int n) {
    for (int i = 0; i < n-1; ++i) {
        int k = i;
        for (int j = i + 1; j < n; ++j) {
            if (A[j] < A[k])
                k = j;
        }
        if (i != k)
            swap(A[k], A[i]);
    }
}
```

Complexity

- SelectionSort() Question
  - How long does the function take to run
    - Proportional to n*n time units, where n is the number of elements in the list
- General question
  - How fast can we sort using the perfect comparison-based method
    - The best possible worst case time is proportional to n log n time units

Multi-Dimensional Arrays

- Syntax

  `btype mdarray[size_1][size_2] ... [size_k]`

- Where
  - k - dimensional array
  - **`mdarray`**: array identifier
  - **`size_i`**: a positive constant expression
  - **`btype`**: standard type or a previously defined user type and is the base type of the array elements

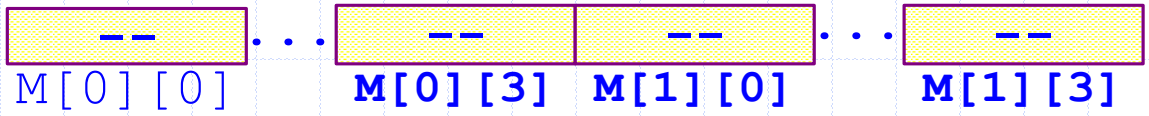
- Semantics
  - **`mdarray`** is an object whose elements are indexed by a sequence of **`k`** subscripts
  - the **`i`**-th subscript is in the range **`0 ... size_i - 1`**

Memory Layout

- Multidimensional arrays are laid out in row-major order
- Consider

```
int M[2][4];
```

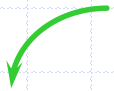
- `M` is two-dimensional array that consists of 2 subarrays each with 4 elements.
  - 2 rows of 4 elements
- The array is assigned to a contiguous section of memory
  - The first row occupies the first portion
  - The second row occupies the second portion



Identity Matrix Initialization

```
const int MaxSize = 25;
float A[MaxSize][MaxSize];
int nr = PromptAndRead();
int nc = PromptAndRead();
assert((nr <= MaxSize) && (nc <= MaxSize));
for (int r = 0; r < nr; ++r) {
   for (int c = 0; c < nc; ++c) {
      A[r][c] = 0;
   }
   A[r][r] = 1;
}
```

Matrix Addition Solution



Notice only first brackets are empty

```
void MatrixAdd(const float A[][MaxCols],
 const float B[][MaxCols], float C[][MaxCols],
 int m, int n) {
   for (int r = 0; r < m; ++r {
      for (int c = 0; c < n; ++c) {
         C[r][c] = A[r][c] + B[r][c];
      }
   }
}
```