Introduction to Arrays

- Array definition:
  - A collection of data of same type
- Another aggregate data type
  - Means "grouping"
    - int, float, double, char are simple data types
- Used for lists of like items
  - Ex: scores, temperatures, names, etc.
  - Avoids declaring multiple simple variables
  - Can manipulate "list" as one entity

Declaring Arrays

- Declare the array \( \Rightarrow \) allocates memory
  
  Ex: int score[4];
  - Declares array of 4 integers named "score"
  - Similar to declaring four variables:
    - score[0], score[1], score[2], score[3];

- Individual parts called many things:
  - indexed (or subscripted) variables
  - "elements" of the array
  - value in brackets called index or subscript
  - index ranges from 0 to (size-1) \( \Rightarrow \) must be one integer
Accessing Arrays

- Access using index/subscript
  ```cpp
cout << score[3];
  ```
- Notice two uses of brackets:
  - in declaration, specifies SIZE of one array
  - Anywhere else, specifies an index (subscript)
- Size and index need not be literal, can be a variable
  ```cpp
  int score[MAX_SCORES];
  ```
  - `MAX_SCORES` is a variable during declaration
  - `score[n+1] = 99;` identical to: `score[3]`

Potential Usage of Array

- Powerful storage mechanism
  - Can issue commands like:
    - "Do this to $i$th indexed variable" where $i$ is computed by program
    - "Display all elements of array score"
    - "Fill elements of array score from user input"
    - "Find highest value in array score"
    - "Find lowest value in array score"

Example of Using an Array

Problem: input 4 scores and compute the deviation from the maximum of 4.

```cpp
int score[4], max; 
cout << “Enter 4 scores:
”; cin >> score[0];
max = score[0];
for (int idx=1; idx<4; i++)
{ 
cin >> score[idx];
if (score[idx] > max)
{ 
  max = score[idx];
}
}
```

Use for Loops with Arrays

- Natural counting loop
  - Naturally works well “counting through“ elements of an array
- Example:
  ```cpp
  for (int idx=0; idx<4; idx++)
  { 
cout << “Max score = “ << max << endl;
  cout << “Differences are: \n”;
  for (int idx=0; idx<4; idx++)
  { 
cout << score[idx];
  cout << “ off by “;
  cout << (max-score[idx]) << endl;
  }
  }
  ```
  - loop control variable `idx` counts from `{0...4}`
Major Array Pitfall

- Array indexes always start with zero!
- Zero is "first" number to computer scientists
- C++ allows you go beyond range
  Ex:
  ```cpp
  int score[4];
  score[10] = 5;
  ⇒ unpredictable results
  ⇒ compiler will not detect/warn these errors!
- Programmers are free to "stay in range“ or not

Example of Major Array Pitfall

- Indexes range from 0 to (array_size-1)
  - Example:
    ```cpp
    double temperature[24];
    // 24 is array size
    // temperature: an array of 24 double values
    - They are indexed as: temperature[0],
      temperature[1], ..., temperature[23]
    - Common mistake: temperature[24] = 5;
      - index 24 is "out of range"!
      - no warning, possibly disastrous results

Defined Constant as Array Size

- Always use defined/named constant for array size, not one number
  - improves readability
  - improves versatility
  - improves maintainability
- Example:
  ```cpp
  const int NUM_STUDENTS = 4;
  int score[NUM_STUDENTS];
  ```

Uses of Defined Constant

- Use everywhere size of array is needed
  - In for loop for traversal:
    ```cpp
    for (idx=0; idx<NUM_STUDENTS; idx++) {
      // Manipulate array
    }
    ```
  - In calculations involving size:
    ```cpp
    iLastIndex = (NUM_STUDENTS - 1);
    ```
  - When passing array to functions (later)
    - If the size changes ⇒ requires only ONE change in your program!
Arrays in Memory

- Recall simple variables:
  - Allocated memory in an "address"
- Array declarations allocate memory for entire array
- Sequentially-allocated
  - Means addresses allocated "back-to-back"
  - Allows indexing calculations
    - Simple "addition" from array beginning (index 0)

Example:

```c
short int a[6]; char v1; ...
  int v2;
```

- `short int` takes 2 bytes for each
- `v1` takes 1 byte
- `v2` takes 4 bytes

An Array in Memory

- **Address**
- **Data**
  - `a[0]`
  - `a[1]`
  - `a[2]`
  - `v1`
  - `v2`
  - `unused`

Initializing Arrays

- As simple variables can be initialized at declaration:
  ```c
  int price = 0; // 0 is initial value
  ```
- Arrays can as well:
  ```c
  int children[3] = { 2, 12, 1 };
  ```
- Equivalent to following:
  ```c
  int children[3];
  children[0] = 2;
  children[1] = 12;
  children[2] = 1;
  ```

Auto-Initializing Arrays

- If fewer values than size supplied:
  - Fills from beginning
  - Fills "rest" with zero of array base type
  ```c
  long vec[15] = {-1};
  // vec[0]=-1, vec[1]=0 ... vec[14]=0
  ```
- If array-size is left out:
  - Declares array with size required based on number of initialization values
  ```c
  int b[] = {5, 12, 11};
  ```
  - Allocate array `b` to have the size of 3
**Special: char Array**

- char array: an array consists of multiple characters

  - Example:
    ```
    char name[0] = 'W';
    char name[1] = 'e';
    char name[2] = 'n';
    char name[3] = '\0';
    ```

- Will be introduced in details in the later lecture

**Arrays in Functions**

- Pass **indexed variables** to functions
  - array elements as arguments
    - Ex: `func(c[0],...)`
    - an individual "element" of an array can be function parameter
  - Pass **entire arrays** to functions
    - array names as arguments
    - Ex: `func(arrayName,...)`
    - all array elements can be passed as one entity
  - As **return value** from function
    - can be done and discussed in later lecture

**Indexed Variables as Arguments**

- Indexed variable handled same as simple variable of array base type

  - Given this function declaration:
    ```
    void myFunction(double par1);
    ```
  - And these declarations:
    ```
    int idx; double n, a[10];
    ```
  - Can make these function calls:
    ```
    myFunction(idx); // idx is converted to double
    myFunction(a[3]); // a[3] is double
    myFunction(n); // n is double
    ```

**Subtlety of Indexing**

- Consider the example:
  ```
  myFunction(a[idx]);
  ```
  - Value of `idx` is determined first
    - `idx` determines which indexed variable is sent
  - More example
    ```
    myFunction(a[idx*5]);
    ```
    - Perfectly legal, from compiler's view
    - Programmer responsible for staying "in-bounds" of array
    - mean checking if `idx` falls in the legal range
Entire Arrays as Arguments

- Formal parameter can be entire array
  - Argument then passed in function call is array name
  - Called "array parameter"
- Send size of array as well
  - Typically done as second parameter
  - Simple int type formal parameter

Example of Arrays as Arguments

```c++
int fillUp(int a[], int size) {
    cout << "Enter " << size << "numbers:\n";
    for (int idx=0; idx<size; idx++) cin >> a[idx];
    cout << "The last array index is ";
    cout << (size-1) << endl;
}
```

In main() function definition, consider the following calls:
```c++
int score[4], numberOfScores = 4; fillup(score, numberOfScores);
```
- 1st argument score is entire array
- 2nd argument numberOfScores is an integer
- Note no brackets in array argument!

Arrays as Arguments: How?

- What's really passed?
- Think of array as 3 "pieces"
  - address of first indexed variable (arrName[0])
  - array base type
  - size of array
- Only 1st piece is passed!
  - Just the beginning address of array
  - Very similar to "pass-by-reference"

Array Parameters

- May seem strange
  - No brackets in array argument when being called
  - Must send size separately!!
- One nice property:
  - Can use SAME function to fill any size array (of same type)!
  - Exemplifies "re-use" properties of functions
  - Example:
    ```c++
    int score[4], time[10];
    fillUp(score, 4);
    fillUp(time, 10);
    ```
**const Parameter Modifier**

- Recall: array parameter actually passes address of 1st element
  - Similar to pass-by-reference
- Function can then modify array!
  - Often desirable, sometimes not!
- Protect array contents from modification
  - Use "const" modifier before array parameter
  - Called "constant array parameter"
  - Tells compiler to "not allow" modifications
  - Ex: int fillUp(const int a[], int size)

**Functions that Return an Array**

- Functions cannot return arrays same way simple types are returned
- Requires use of a "pointer"
- Will be discussed in later lectures...

**Programming with Arrays**

- Plenty of uses
  - Partially-filled arrays
    - must be declared some "max size"
  - Sorting
  - Searching

**Partially-Filled Arrays**

- Difficult to know exact array size needed
  - Ex: The maximum number of students in one class is 50 but may be 45 or 47 in fact
- Must declare to be largest possible size
  - Must then keep "track" of valid data in array
  - Additional "tracking" variable needed
  - int numberUsed;
  - Tracks current number of elements in array
#include <iostream>
using namespace std;
const int MAX_NUM_SCORES = 100;
void initializeArray(int a[], int size);
void fillArray(int a[], int size, int& usedSize);
double computeAverage(const int a[], int usedSize);
int main()
{
    int score[MAX_NUM_SCORES], numUsed = 0;
    initializeArray(score, MAX_NUM_SCORES);
    fillArray(score, MAX_NUM_SCORES, numUsed);
    double avg;
    avg = computeAverage(score, numUsed);
    return 0;
}

// called from initializeArray(score, MAX_NUM_SCORES);
void initializeArray(int a[], int size)
{
    // fill up cells a[0] ... a[size-1] with -1
    for (int idx = 0; idx < size; idx++)
    {
        a[idx] = -1;
    }
}

// called from fillArray(score, MAX_NUM_SCORES, numUsed);
int fillArray(int a[], int size, int& usedSize)
{
    cout << "Enter up to " << size << " numbers:\n";
    int next = -1, idx = 0;
    cin >> next;
    while ((next >= 0) && (idx < size))
    {
        a[idx] = next;
        idx++;
        cin >> next;
    }
    usedSize = idx;
    return 0;
}
Example of Partially-Filled Arrays (5/6)

```cpp
#include <iostream>
using namespace std;
const int MAX_NUM_SCORES = 100;
void initializeArray(int a[], int size);
void fillArray(int a[], int size, int& usedSize);
double main()
{
    int score[MAX_NUM_SCORES], numUsed = 0;
    initializeArray(score, MAX_NUM_SCORES);
    fillArray(score, MAX_NUM_SCORES, numUsed);
    double avg = computeAverage(score, numUsed);
    return 0;
}
```

Example of Partially-Filled Arrays (6/6)

```cpp
// called from computeAverage(score, numUsed);
double computeAverage(const int a[], int usedSize)
{
    double total = 0.0;
    for (int idx = 0; idx < usedSize; idx++)
        total = total + a[idx];
    if (usedSize >= 0)
    {
        return (total / usedSize);
    }
    else
    {
        cout << "ERROR: no element in array\n";
        return 0;
    }
}
```

Global Constants vs. Parameters

- Constants typically made "global"
  - Declared above main()

- Functions then have scope to array size constant
  - No need to send as parameter then?
    ⇒ Technically yes

- Why should we anyway?
  ⇒ Function definition might be in separate file and function might be used by other programs!

Searching an Array

- Very typical use of arrays ⇒ return the index
  ```cpp
  // called from SearchArray(score, numUsed, target);
  int SearchArray(const int a[], int usedSize, int target)
  {
      int foundIdx = -1;
      for (int idx = 0; idx < usedSize; idx++)
      {
          if (target == a[idx])
          {
              foundIdx = idx;
              break;
          }
      }
      return foundIdx;
  }
  ```
Sorting an Array

- **Input:**
  an array \( c = \{ c[0], c[1], \ldots, c[n-1] \} \) of \( n \) numbers

- **Output:**
  a permutation \( c'[0], c'[1], \ldots, c'[n-1] \) of the input sequence such that
  \( c'[0] \leq c'[1] \leq \ldots \leq c'[n-1] \)

- The number that we wish to sort are known as the **keys**.

**Example:**
- Input: 8 2 4 9 3 6
- Output: 2 3 4 6 8 9

Example of Insertion Sort

**IDEA** sort your poker cards

![Insertion Sort Example](image)

Pseudocode of Insertion Sort

Insertion-Sort(\( A \))

1. for \( j \leftarrow 2 \) to length[\( A \)]
2. do key \( \leftarrow A[j] \)
3. //Insert \( A[j] \) into the sorted sequence \( A[1..j-1] \)
4. \( i \leftarrow j - 1 \)
5. while \( i \geq 0 \) and \( A[i] > key \)
6. do \( A[i+1] \leftarrow A[i] \)
7. \( i \leftarrow i - 1 \)
8. \( A[i+1] \leftarrow key \)

Multi-dimensional Arrays

- Declare arrays with more than one index
  - char page[30][100]; //fixed sizes for two dimensions
- Two indexes: An "array of arrays"
- Visualize as:

```
<table>
<thead>
<tr>
<th>page[0][0]</th>
<th>page[0][1]</th>
<th>...</th>
<th>page[0][99]</th>
</tr>
</thead>
<tbody>
<tr>
<td>page[1][0]</td>
<td>page[1][1]</td>
<td>...</td>
<td>page[1][99]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>page[29][0]</td>
<td>page[29][1]</td>
<td>...</td>
<td>page[29][99]</td>
</tr>
</tbody>
</table>
```

- C++ allows any number of indexes
  - Typically no more than two levels
Initializing Multi-dimensional Arrays

- Unless specified, all initial values of arrays are garbage.
- You can specify initial values by enclosing each row in curly braces like this
  ```
  char ticTacToeBoard[3][3] =
  { {'O', 'X', 'X'},
    {'O', 'O', 'X'},
    {' ', 'X', ' '} }
  ```

Multi-dimensional Array Parameters

- Similar to one-dimensional array
  - 1st dimension size not given: provided as second parameter
  - 2nd dimension size is given ⇒ remember
- Example:
  ```
  void DisplayPage(const char p[][100], int Dim1)
  {
    for (int idx1=0; idx1<Dim1; idx1++)
      for (int idx2=0; idx2<100; idx2++)
        cout << p[idx1][idx2];
      cout << endl;
  }
  ```

Summary (1/2)

- Array is collection of "same type" data
- Indexed variables of array used just like any other simple variables
- for-loop "natural" way to traverse arrays
- Programmer responsible for staying "in bounds" of array
- Array parameter is "new" kind
  - Similar to call-by-reference

Summary (2/2)

- Array elements stored sequentially
  - "Contiguous" portion of memory
  - Only address of 1st element is passed to functions
- Partially-filled arrays ⇒ more tracking
- Constant array parameters
  - Prevent modification of array contents
- Multi-dimensional arrays
  - Create "array of arrays"