Arrays

One-Dimensional Arrays

• An array is a homogeneous (all components are of the same DataType) data structure

• A component of an array is accessed by its position in the data structure - not by an identifier

• N-dimensional array - a structured collection of components, all of the same type, that is given a single name\(N = 1, 2, 3, \ldots\)

• Syntax Template: \texttt{ArrayDeclaration}\n  \begin{verbatim}
  
  DataType ArrayName [ConstIntExpression] [ConstIntExpression]… ;
  
  \end{verbatim}

  • DataType describes what is stored in each component of the array
  • ConstIntExpression is an integer expression consisting of literal or named constants only. This expression specifies the number of components for a dimension in the array. Must be an integer value and greater than 0
Accessing Individual Components

• The index value for each dimension in an array starts at 0 and goes to n-1 where n is the number of components for that dimension
  • int Array[4] – one dimensional array with 4 components – index values go from 0 to 3
  • int Array[4][6] – two dimensional array with 24 components – index values go from 0 to 3 for the rows and 0 to 5 for the columns.

• To access an individual array component, the array name is followed by an expression enclosed in brackets
  ArrayName[IndexExpression]
  IndexExpression may be a constant, a variable, or a combination of variables, operators and function calls as long as the resulting DataType is an integer.

• Each array component can be treated exactly the same as any simple variable of the same DataType

```
int speed[10]; // one-dimensional array of 10 integer elements
const int MAX_ATTEMPT = 10;
float player[MAX_ATTEMPT]; // an array of 10 float elements
float player2[MAX_ATTEMPT*2]; // array of 20 float elements
int j;
float value;
... // other statements

player[0] = 34.5;
// assign 55 to array element specified
player[(j*4 + 5)%MAX_ATTEMPT] = 55.0;
value = 5*player[4]+3.0;
// initialize all array elements of player to 0.0
for (j = 0; j < MAX_ATTEMPT; j++)
    player[j] = 0.0;
```

See Program Arrays_01.cpp for examples
Out-of-Bounds Array Indices

• An index value in C++ ranges from 0 to the array size minus 1
• C++ does not check for invalid (out-of-bounds) array indexes during compile time or at run time
• Reference page 387-389 of text for the assert Library Function
  • Can write an executable assertion to verify that the index of an array is within the proper range.
  • Very useful if the index value is a calculated value
    #include <cassert>
    ... 
    index = someInt*4/someValue + 4;
    assert (index >= 0 && index < array_size);
    array[index] = array[index] + 1;
    ...

    See Program Arrays_02.cpp for examples

Initialization of Arrays

• Arrays can be initialized in the program or during declaration

• Can omit the size when initializing during declaration. The number of values determine the array size (one-dimensional arrays)
  float change[ ] = { 0.01, 0.05, 0.1, 0.25 };

• Array entries must be accessed individually
Aggregate Operation on Arrays

- **Aggregate Array Operations** - The **ONLY** operation that can be performed on an array as a whole is pass it as an argument to a function. The following are all INVALID aggregate operations:

  For arrays: int X[50]; int Y[50];
  - No aggregate assignment of arrays: X = Y;
  - No aggregate comparison of arrays: if (X == Y)
  - No aggregate I/O for arrays: cout << X;
  - No aggregate arithmetic: X = X + Y;
  - Cannot return an array as the value of a value-returning function:
    return X; // invalid aggregate array operation

Passing Arrays as Arguments

- By default, arrays are always passed by reference. Therefore, you never need to use the & when declaring an array as a parameter.

- In the parameter list of a function, the declaration of a one-dimensional array does not include a size within the brackets - more on this later. If a size is included, it is ignored.

```c++
void init_array (int[ ], int); // prototype
...
int main ()
{
    int score[25];
    ...
    init_array(score, 25); // function call
}
void init_array (int ray[ ], int num_cells) // function definition
{
    for (int j = 0; j < num_cells; j++)
        ray[j] = 0;
}
Passing Arrays as Arguments(continued)

- To pass an array such that the function cannot modify it, use the reserved word `const` in the declaration of the parameter. The use of `const` does not affect the function call format.
  ```c
  void write (const float[], int); // function prototype
  void write (const float ray[], int size) // function defin.
  {
  ...
  }
  ```
- It is a common mistake to try and pass a single array element to a function instead of the array.
  ```c
  float test[20]; // Array declaration
  ...
  write (test, 20); // proper way to call the function with the array
  ```
  ```c
  write (test[15], 20); // tries to pass a single array element - test[15]
  // This line results in a compile error
  ```
  See Program Arrays_04.cpp for examples

Passing Arrays as Arguments(continued)

- When an array is passed as an argument in a function call, the `base address` – the memory address of the first element of the array (index value of 0) – is sent to the function.
  - Tells the function where the array resides in memory
  - Allows the function to correctly access any element of the array
Using Typedef with Arrays

- The Typedef statement provides a way to give an additional name to an existing DataType

- An example with arrays:
  - \texttt{typedef int IntArray[50];}
  - This statement \texttt{defines} a new DataType \texttt{IntArray}. Variables declared of DataType \texttt{IntArray} will be a one-dimensional 50-element array of type int
  - \texttt{IntArray position; // is the same as: int position[50];}

- More useful with multi-dimensional arrays

  \texttt{See program Arrays\_05.cpp for an example of this concept}

Arrays of Records (structs)

```
const int MAX_RECORDS = 50;
struct MachineRec
{
    string mach_name;
    int id;
    float speeds[3];
};
int main ()
{
    MachineRec MachLog[MAX_RECORDS];
    int num;
}
```

\texttt{See programs Arrays\_06.cpp and Arrays\_07.cpp for this example}
Arrays of Records (cont)

- In this example MachLog is an array of structures.
  - MachLog[4].id
    - specifies the id member of the 5th structure in the array MachLog
    - specifies the third value of the speeds member of the 16th structure in the array MachLog

Miscellaneous Array Information

- Since array indexing starts at 0, my_array[5] indicates the 6th component of the array - not the 5th. (0, 1, 2, 3, 4, 5, …)
- **Subarray processing:**
  - to avoid processing empty components in an array, it is beneficial to keep track of how many components are actually filled.
  - This processing requires that the components are filled in a specified order (by row, by column or by square amount – same number of rows and columns) and not randomly
Two-Dimensional Arrays

- **Two-Dimensional Array**: A collection of components, all of the same type, structured in two dimensions (rows and columns). Each component is accessed by a pair of indexes.

  ```
  const int NUM_ROWS = 10;
  const int NUM_COLS = 10;
  float my_array[NUM_ROWS][NUM_COLS];
  Float arr[][3] = { {1.0, 2.0, 3.0}, {3, 2, 1}};
  my_array[4][8] = 4.5;
  ```

- **SyntaxTemplate**: ArrayComponentAccess – 2 dimensions
  - `ArrayName [IndexExpression][IndexExpression]`
  - `IndexExpression` must result in an integer value

Initializing Two-Dimensional Arrays

- Easiest way is to use nested loops
  ```
  for (int row = 0; row < NUM_ROWS; row++)
    for (int col = 0; col < NUM_COLS; col++)
      my_array[row][col] = 0.0;
  ```

- Could use a single loop and calculate the row and column index determined from the single loop value
  ```
  for (int loop=0; loop < NUM_ROWS*NUM_COLS; loop++)
  {
    row = loop/NUM_COLS; // int math
    col = loop - row*NUM_COLS;
    my_array[row][col] = 0.0;
  }
  ```

  See program Arrays_08.cpp for an example
Processing Two-Dimensional Arrays

• Generally access the array in one of four patterns - randomly, along rows, along columns or throughout the entire array

• Sub-array processing may be required for any of the above methods

---

Sum the Rows example - assume all variables declared correctly

```c
for (row = 0 ; row < rowsFilled; row++)
{
    total = 0;
    for (col = 0; col < colsFilled; col++)
        total = total + arr[row][col];
    cout << “Row sum: “ << total << endl;
}
```

• Sum the Columns has the same format as above. Switch the for statements for row and col, and change Row to Column in cout
Passing Two-Dimensional Arrays as Arguments

- Passing a two-dimensional array allows you to omit the size of the first dimension, but not the second
  - In memory multi-dimensional arrays are arranged in one long line of memory cells
  - Functions receiving the base address of an array need to know how many columns (therefore how many components in each row) are in the array

```
void MyFunc( float alpha [[10]]; // prototype
```

See Program Arrays_09.cpp for an example

More on Two-Dimensional Arrays

- Using a typedef statement (pages 537, 553, 563, 565) with two-dimensional arrays solves/simplifies problems with
  - Making changes to the array indexes when the size changes
  - Mismatches in argument and parameter sizes

```
const int NUM_ROW = 10; // global constant
const int NUM_COL = 15; // global constant
typedef int ArrayType[NUM_ROW][NUM_COL]; // global definition
void process_array(ArrayType ); // function prototype
int main()
{
    ArrayType MyArray; // declare MyArray to be of type ArrayType
    ... // statements in main go here, and below main the function definition
    process_array(MyArray); // function call
}
```

See Program Arrays_09.cpp for an example
Multi-dimensional Arrays

- Expand the two-dimensional concept out to as many dimensions as required.
  - Using enumerator types may help with understanding the array dimensions in the program
  - Using integer constants for the dimension values will help with understanding the dimensions when accessing, processing and writing code for the arrays.
  - Using a typedef statement for the array type will help avoid problems with mismatched sizes.
- Read the Case studies
- Read Testing and debugging on pages 581 - 584