ARRAYS

OVERVIEW

OVERVIEW

- **In many programs, we need to store and process a lot of data with the same data type**
  - Processing test scores to find the class average
  - Tabulating bank deposits and withdrawals
  - Displaying images on a computer screen

- **Arrays in C++ give us a way to accomplish this goal**
  - Declare an array of desired data type and size
  - Store data values in each array location
  - Process data values to solve a specific problem

OVERVIEW

- **How do we process data in arrays?**
  - Depends on needs of the application
  - Some applications create summaries of data in array
  - Some applications print a subset of data in array
  - Some applications search for data in the array
  - Some applications move data around in the array

- **How to we implement this array processing?**
  - Use iteration to loop over array elements
  - Use functions to simplify code reuse

OVERVIEW

- **We need to learn a variety of array processing algorithms in order to become strong C++ programmers**

- **Each array processing algorithm has its pros/cons**
  - Some have faster run times, some are slower
  - Some take more memory, some take less memory
  - Some are complex to implement, some are simple

- **To understand these differences, we must learn scientific methods for algorithm analysis and program testing**

OVERVIEW

- **Lesson objectives:**
  - Learn the syntax for declaring arrays in C++
  - Learn how to store and process data in arrays
  - Learn how to search and sort data in arrays
  - Study example programs showing their use
  - Complete online labs on arrays
  - Complete programming project using arrays

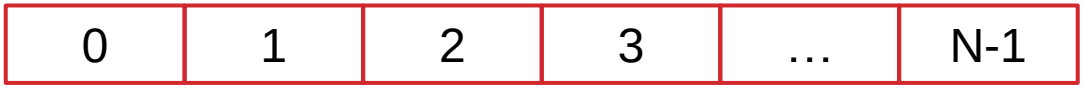
ARRAYS

PART 1

ARRAY BASICS

DECLARING ARRAYS

- **Arrays were invented to conveniently store multiple values of the same data type in one variable**
  - Picture an array as a long box divided into N slots



- **Array elements are stored in separate memory locations and accessed based on their position**
  - The first array element is in location 0
  - The last array element is in location N-1

DECLARING ARRAYS

- **The syntax for an array declaration is:**

  data_type array_name [array_size];

  where

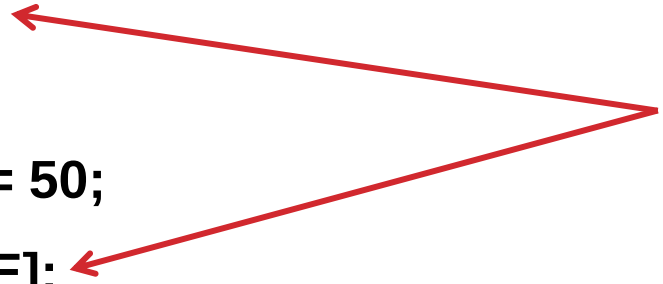
  data_type can be any basic C++ type (int, float, etc.)

  array_name follows C++ variable name rules

  array_size is an integer or integer constant

DECLARING ARRAYS

```
// Valid array declarations
float Data[100];
int List[20];
const int SIZE = 50;
char Name[SIZE];
```



The array_size can be specified by an integer or an integer constant

DECLARING ARRAYS

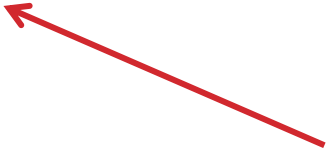
```
// Dynamic array declaration
int Size = 0;
while (Size <= 0)
{
   cout << "Enter array size: ";
   cin >> Size;
}
int Array[Size];
```

Using an integer variable for array size works on some but not all C++ compilers

DECLARING ARRAYS

```
// Invalid array declarations
int Length = -1;
float Data[Length];
int List[31.75];
```

Array size must be a positive integer



You can not use floats to specify the array size

DECLARING ARRAYS

- **The total number of bytes in memory for an array is given by (number of elements in array) * (number of bytes for each element)**

float Data[100];

- One float takes 4 bytes
- Array size is 100 * 4 = 400 bytes

char name[20];

- One char takes 1 byte
- Array size is 20 * 1 = 20 bytes

ARRAY ACCESS

- **To access an array element, we need to give name of variable and the index (location) of desired element**
  - Eg: array_name[array_index]

- **In C++ arrays are always "zero indexed"**
  - The first array element is at location 0
  - The last array element is at location N-1
  - If you attempt to use an array index outside the range 0..N-1 you will get an error when your program is running

ARRAY ACCESS

```
// Valid array access
const int SIZE = 100;
float Data[SIZE];
…
Data[0] = 7;
Total = Total + Data[2];
cout << Data[7];
```

We use variables in an array just like any other variable, as long as the array index is within the range 0..SIZE-1

ARRAY ACCESS

```
// Invalid array access
Data[4.3] = 28;
…
Data[-8] = 0.0;
Data[200] ++;
…
cin >> Data;
cout << Data;
```

The array index can not be a float

Run time errors may occur if array index is outside 0..99 range

We can not read or write a whole array at one time

ARRAY INITIALIZATION

- **It is very important to initialize arrays before use**
  - Using an uninitialized variable can cause major bugs
  - Arrays are supposed to be initialized to 0 by default
  - Sadly, this is not true for all C++ compilers, so we should always do array initialization ourselves to be safe

- **We can store initial values in an array at declaration time**
  - Give collection of N values to store in array of size N
  - If fewer than N values are given, the rest are set to 0
  - Assign an array of same size and data type

ARRAY INITIALIZATION

```
// Valid array initialization
const int SIZE = 10;
int Value[SIZE] = {3,1,4,1,5,9,2,6,5,3};
int Copy[SIZE] = Value;
…
char Name[SIZE] = {'J', 'O', 'H', 'N'};
...
float Scores[] = {93.5, 92.0, 90.1, 85.7, 83.3, 76.5};
```

The rest of this character array is initialized to 0 (the null character)

Size of this array is determined by number of values to right (6)

ARRAY INITIALIZATION

```
// Invalid array initialization
float Data[20] = Value;
…
int Numbers[5] = {2,1,4,1,5,1,6};
```

The Value array is a different size (10) so it can not be used to initialize the Data array

The number of initialization values can not be larger than the array size

ARRAYS AND LOOPS

- **It is very natural to use loops to process arrays**
  - Read N input values into an array
  - Write N output values from an array
  - Calculate total of N values in array

- **We must take care to stay within array bounds**
  - Never use index less than 0
  - Never use index greater than N-1
  - If you do go outside this 0..N-1 range, it may cause a "memory segmentation fault" error at run time

ARRAYS AND LOOPS

```
// Input output example
int Data[10];

for (int i = 0; i < 10; i++)
   cin >> Data[i];

for (int i = 0; i < 10; i++)
   cout << Data[9-i];
```

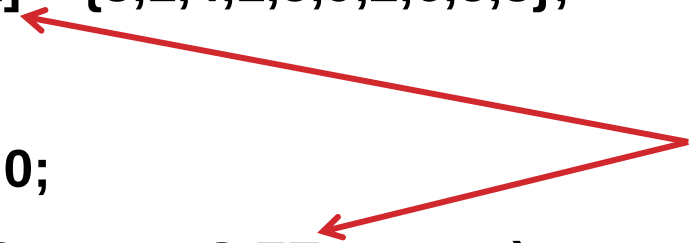

Loop to read 10 values into the Data array

Loop to write 10 Data values in reverse order

ARRAYS AND LOOPS

```
// Average calculation example
const int SIZE = 10;
int Value[SIZE] = {3,1,4,1,5,9,2,6,5,3};

float Total = 0.0;
for (int pos = 0; pos < SIZE; pos++)
   Total = Total + Value[pos];
float Average = Total / SIZE;
```



Here we use the SIZE constant in the array declaration and also in the array processing loop

USING PARTIAL ARRAYS

- **What happens if we do not know array size in advance?**
  - It is possible but tricky to allocate dynamic arrays
  - Easier to declare a large array and use only part of it

- **How do we do this?**
  - We guess the maximum size needed for the array
  - Declare the array to be the maximum size needed
  - We use only part of this array to store our data
  - We also keep track of how much of the array is currently being used in a “Count” variable

USING PARTIAL ARRAYS

- **Example: Reading student grades into an array**
  - Assume the user knows how many grades they will enter
  - Prompt the user for the grade count
  - Read the grade count from the user
  - Loop reading grades into array
  - Process the grade array in some way

- **Sample input:**

  5

  78 85 91 88 94

USING PARTIAL ARRAYS

```
// User enters array count followed by grades
const int SIZE = 1000;
float Grades[SIZE];
```

We are guessing that the user waill never want to use more than 1000 values

```
int Count = 0;
cout << "Enter count: ";
cin >> Count;
```

We ask the user for how much of the array they want to use today

USING PARTIAL ARRAYS

```
if (Count > SIZE)
   Count = SIZE;
```

We do error checking to make sure Count <= 1000

```
for (int i = 0; i < Count; i++)
{
     cout << "Enter grade: ";
     cin >> Grade[i];
}
```

We can now loop from 0 up to Count-1 reading data from the user into the array

USING PARTIAL ARRAYS

- **Example: Reading student grades into an array**
  - Assume the count is not known in advance and the grade data will be followed by a sentinal value of -1
  - Read first data value from user
  - While data value is not the sentinal value
    - Store the grade in array
    - Read the next data value from user
  - Process the grade array in some way

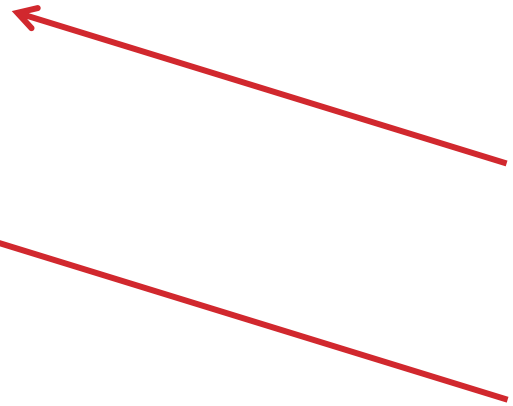
- **Sample input:**

  78 85 91 88 94 -1

USING PARTIAL ARRAYS

```
const int SIZE = 1000;
float Grade[SIZE];

int Count = 0;
float Input = 0.0;
```

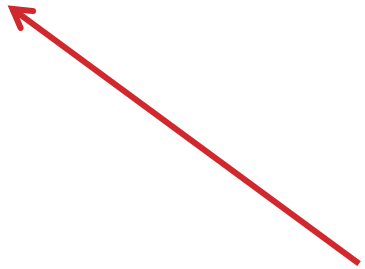


We declare an array that can hold up to 1000 grades

We use Count to tell us how many were actually read

USING PARTIAL ARRAYS

```
// User enters grades followed by -1 sentinel value
cout << "Enter grade: ";
cin >> Input;
while ((Input != -1) && (Count < SIZE))
{
   Grade[Count] = Input;
   Count = Count + 1;
   cout << "Enter grade: ";
   cin >> Input;
}
```



We stop reading user input when 1000 values are entered or when the user types -1 sentinel value

ARRAYS AS PARAMETERS

- **How do we declare array parameters?**
  - Add the characters [ ] after the array name in the parameter declaration to tell compiler this is an array
- **How do we use array parameters?**
  - Just give the name of the array in the function call
- **What type of parameter is this?**
  - Arrays are automatically treated as reference parameters
  - There is no way to pass entire arrays as value parameters
  - We can add the keyword const before the data type to make the array parameter read only

ARRAYS AS PARAMETERS

- **How can we add one to all values in an array?**

- **Write a function to process the array**
  - Declare array parameter
  - Declare array size parameter
  - Loop over array in function doing operation

- **Call this function in the main program**
  - Pass in the name of the array
  - Pass in the size of the array

ARRAYS AS PARAMETERS

```
// Declare function to increment all values in an array
void AddOne( const int Size, float Value[ ] )
{
   for (int i= 0; i < Size; i++)
      Value[i] = Value[i] + 1.0;
}
```

We can call this function with float arrays of any size

We use the Size parameter to tell function how large the input array is

We loop over all of the elements of the array and add 1.0 to each value

ARRAYS AS PARAMETERS

```
// Call function to process array
float Data[10] = {1,2,3,4,5,6,7,8,9,10};
AddOne( 10, Data );
```

The first parameter is the size of the input array “Data”

The second parameter is the name of the array we want to process “Data”

ARRAYS AS PARAMETERS

- **How can copy data values from one array to another?**

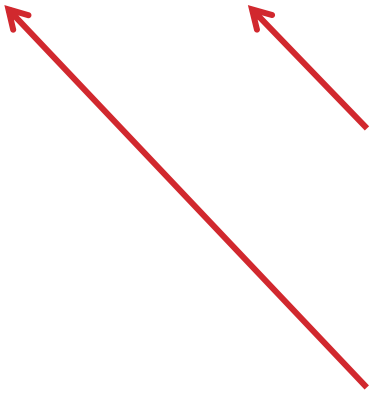
- **Write a function to process the arrays**
  - Declare two array parameters
  - Declare array size parameter
  - Loop over array in function doing operation

- **Call this function in the main program**
  - Pass in the names of the arrays
  - Pass in the size of the arrays

ARRAYS AS PARAMETERS

```
// Declare function to copy array values
void CopyData( const float In[ ], float Out[ ], const int Size )
{
   for (int i=0; i < Size; i++)
      Out[i] = In[i];
}
```

We loop over all of the elements of the “In” array and copy the value to “Out”



The array parameter “Out” can be modified in function

The array parameter “In” is a read only and can not be modified

ARRAYS AS PARAMETERS

```
// Call function to process array
float Data1[10] = {1,2,3,4,5,6,7,8,9,10};
float Data2[10] = {0,0,0,0,0,0,0,0,0,0};
CopyData( Data1, Data2, 10 );
```

This will copy Data1 data into the Data2 array

ARRAYS AS PARAMETERS

```
// Call function to process array
float Data1[10] = {1,2,3,4,5,6,7,8,9,10};
float Data2[10] = {0,0,0,0,0,0,0,0,0,0};
CopyData( Data2, Data1, 10 );
```

This will copy Data2 data into the Data1 array

ARRAYS AS PARAMETERS

```
// Call function to process array
float Data1[10] = {1,2,3,4,5,6,7,8,9,10};
float Data2[10] = {0,0,0,0,0,0,0,0,0,0};
CopyData( 10, Data1, Data2 );
```

This will cause a compiler error because the function parameters are in the wrong order

ARRAYS AS PARAMETERS

```
// Call function to process array
float Data1[10] = {1,2,3,4,5,6,7,8,9,10};
float Data2[10] = {0,0,0,0,0,0,0,0,0,0};
CopyData( Data1, Data2, 20 );
```

This will cause array bounds errors with very strange side effects

SUMMARY

- In this section, we saw how to declare, initialize and access arrays in C++
- We also saw how loops could be used to read/write and process arrays in different ways
- Next, we discussed how arrays can be used to store and process a variable number of elements
- Finally, we showed how arrays can be passed into functions as parameters

ARRAYS

PART 2

ADVANCED ARRAYS

CHARACTER ARRAYS

- **Arrays of characters in C++ are called cstrings**
  - When C was invented, cstrings were the only way to store textual information like names and addresses
  - When C++ was invented, they added the string data type as another way to store textual information

- **C++ treats cstrings differently from other arrays**
  - We can initialize a cstring using: char word[10] = “hello”
  - We can read data into a char array using: cin >> word
  - We can write data from a char array using: cout << word
  - We can use a function library to perform other common operations on arrays of characters

CHARACTER ARRAYS

- **By convention cstrings always end with '\0' null character**
  - If we declare a char array with SIZE=100, we can store up to 99 characters in the cstring
  - When we read data into a cstring, the cin command will automatically add the null char after the user input
  - When we write data from a cstring, the cout command will print all characters before the null char

- **Character and cstring literals**
  - Use single quotes 'a' for character literals
  - Use double quotes "hi mom" for cstring literals

CHARACTER ARRAYS

```
// Input output example
const int SIZE = 10;
char Password[SIZE];
char Name[SIZE] = "Smith";

cout << "Hello Mr. " << Name << endl;
cout << "Enter Password: ";
cin >> Password;
```

Here we declare two cstring variables

Here we use cin and cout to read and write cstring variables

CHARACTER ARRAYS

- **We can include <cstring> for additional string functions**
  - strlen(str) – counts the number of characters that are before the null char in str and returns this value
  - strcpy(str1, str2) – loops over str2 and copies all str2 characters before the null char into str1 (no error checking is done to make sure there is room in str1)
  - strncpy(str1, str2, len) – loops over str2 and copies up to len characters from str2 to str1 (we can perform basic error checking by making len equal to the array size of str1)

CHARACTER ARRAYS

- strcat(str1, str2) – appends a copy of str2 at the end of the str1 parameter (no error checking is done to ensure that there is room in the str1 array)

- strcmp(str1, str2) – loops over str1 and str2 and compares these two strings alphabetically up to the null char and returns an integer code after the comparison

  0 if str1 == str2

  -1 if str1 < str2

  1 if str1 > str2

CHARACTER ARRAYS

```
// Using string functions
cout << "Name length: " << strlen(Name) << endl;
…
// char Copy[SIZE];
strcpy(Copy, Name, SIZE);
…
if (strcmp(Name, Password) == 0)
   cout << "Error:  You can not use name as the password\n";
```

We can not assign cstring variables using Copy = Name

We can not compare cstring variables using Name == Password

CHARACTER ARRAYS

- **Problem 1 – wasted memory space**
  - You need to make the cstring array one character larger than the largest possible string that could be stored
  - You have to decide to either waste memory space or truncate strings when they are stored

- **Problem 2 – potential array bounds problems**
  - It is very easy to go past the array bounds with a cstring by simply reading the user's input "cin >> name"
  - This could potentially overwrite another variable and cause very subtle bugs in the program (hackers love this)

2D ARRAYS

- **In many applications, data can be naturally organized as a two-dimensional grid of values**
  - Data in a spreadsheet
  - Pixels in an image

	A	B	C
1	fred	bob	total
2	3	1	4
3	4	1	5
4	5	9	14
5	2	6	8
6	5	3	8
7			

2D ARRAYS

- **Fortunately C++ (and most other languages) will allow us to define two-dimensional arrays by specifying**
  - The array name
  - The data type
  - The number of rows and columns

```
// Example array declaration
const int ROWS = 5;
const int COLS = 3;
int A[ ROWS ][ COLS ];
```

2D ARRAYS

- **We refer to 2D array locations using [row][column] index**
  - The rows are numbered 0..ROWS-1
  - The columns are numbered 0..COLS-1

← columns →

↓ rows →

A[0][0]	A[0][1]	A[0][2]
A[1][0]	A[1][1]	A[1][2]
A[2][0]	A[2][1]	A[2][2]
A[3][0]	A[3][1]	A[3][2]
A[4][0]	A[4][1]	A[4][2]

2D ARRAYS

- **2D arrays can be initialized much like 1D arrays**
  - We must provide ROWS * COLS values
  - We use curly brackets to group rows of values

int Scores [3][3] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };

1	2	3
4	5	6
7	8	9

2D ARRAYS

- **Declaring 2D array parameters**
  - We use Data[ ][COLS] when declaring a 2D array parameter, where COLS is a predefined constant
  - We do not need to specify the number of ROWS in the 2D array when we declare the Data parameter
  - The number of rows in the 2D array should be given in a separate parameter (this way the function can handle 2D arrays with any number of rows)

- **Passing 2D arrays into functions as parameters**
  - When passing 2D array into a function we just use the name of the array (just like 1D arrays)

2D ARRAYS

- **Consider the problem of storing and displaying characters on an old fashioned VT52 computer terminal**
  - VT52s display 24 rows and 80 columns of characters
  - We need to store these characters in a 2D array

2D ARRAYS

```
// Array declaration
const int ROWS = 24;
const int COLS = 80;
char Screen[ ROWS ][ COLS ];
```

Here we declare a 2D array for the screen

```
// Array initialization
for (int r = 0; r < ROWS ; r++)
   for (int c = 0; c < COLS ; c++)
      Screen[r][c] = ' ';
```

Here we initialize the screen to all spaces

2D ARRAYS

```
// Array parameter declaration
void PrintData(const char Data[ ][ COLS ], int rows )
{
  for (int r = 0; r < rows; r++)
  {
    for (int c = 0; c < COLS; c++)
      cout << Data[r][c];
    cout << endl;
  }
}

// Array parameter usage
PrintData(Screen, ROWS);
```

Here we have a 2D array parameter

Here we loop over the 2D array to print it

2D ARRAYS

- **Consider the problem of processing student grades that are stored in a 2D array, with one row per student, and one column per homework assignment**
  - We can calculate one student's average by totaling the values in one row, and dividing by number of homeworks (the number of columns in the 2D array)
  - We can calculate the class average on one homework by totaling the values in one column, and dividing by the number of students (the number of rows in the 2D array)
  - We can calculate class average on all homework by totaling all the data in the 2D array and dividing by the size of the array (rows * columns)

2D ARRAYS

```
// Array declaration
const int STUDENTS = 40;
const int HOMEWORKS = 15;
float Grades[ STUDENTS ][ HOMEWORKS ];

// Array initialization
for (int r = 0; r < STUDENTS; r++)
   for (int c = 0; c < HOMEWORKS; c++)
      cin >> Grades[r][c];
```

Here we declare a 2D array for the grades

Here we read student scores from user to initialize the 2D arrray

2D ARRAYS

```
// Calculate homework average for one student
int student = 0;
float total = 0.0;
float average = 0.0;
cout << “Enter student index: ”;
cin >> student;
for (int c = 0; c < HOMEWORKS; c++)
    total = total + Grades[student][c];
average = total / HOMEWORKS;
cout << “Average= ” << average << endl;
```

We loop over one row in the 2D array to calculate the homework average for one student in class

2D ARRAYS

```
// Calculate class average for one homework
int homework = 0;
float total = 0.0;
float average = 0.0;
cout << “Enter homework index: ”;
cin >> homework;
for (int r = 0; r < STUDENTS; r++)
    total = total + Grades[r][homework];
average = total / STUDENTS;
cout << “Average= ” << average << endl;
```

We loop over one column in the 2D array to calculate class average for one homework assignment

2D ARRAYS

```
// Calculate class average on all homework
float total = 0.0;
float average = 0.0;
for (int r = 0; r < STUDENTS; r++)
   for (int c = 0; c < HOMEWORKS; c++)
      total = total + Grades[r][c];
average = total / (STUDENTS*HOMEWORKS);
cout << “Average= ” << average << endl;
```

We loop over whole array to calculate class average

SOFTWARE ENGINEERING TIPS

- **Suggestions when using arrays:**
  - Always use constants for the array dimensions
  - Make sure your loops go from 0..N-1
  - Use functions to implement useful array operations

- **Common programming errors:**
  - Invalid array declarations or initializations
  - Array index out of bounds (off by one errors)
  - Missing [ ] in array parameter definitions
  - Attempting to modify a const array parameter

SUMMARY

- In this section, we described how cstrings (arrays of characters) can be used to store and print text
- We also showed how 2D arrays can be defined and used to manipulate two-dimensional data

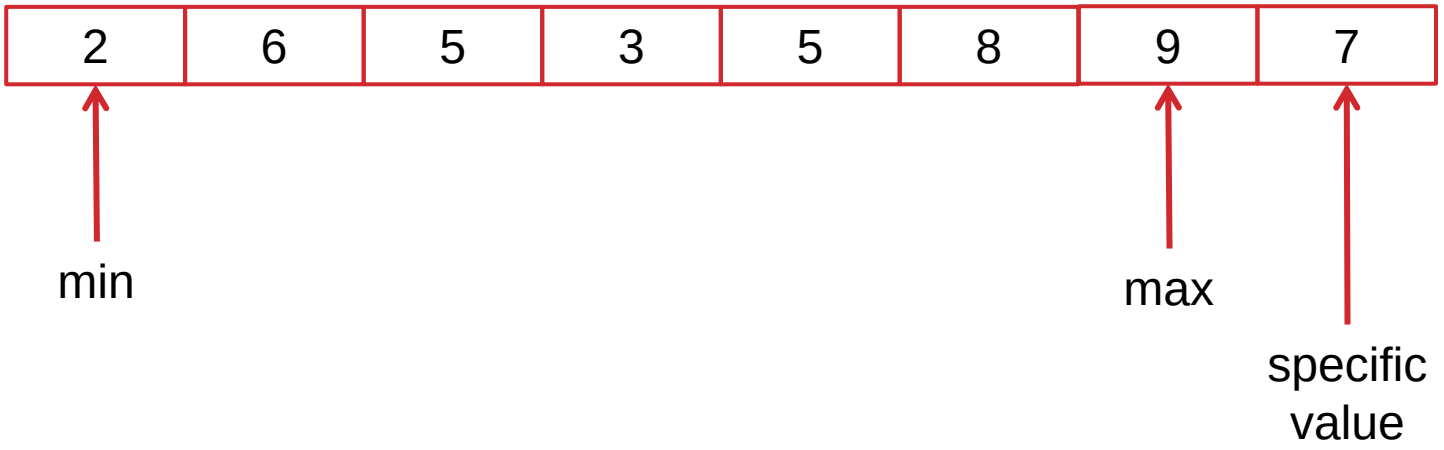
ARRAYS

PART 3

SEARCHING AND SORTING

SEARCHING ARRAYS

- **Once we have stored a collection of values in an array, we can search the array to answer a number of questions:**
  - Does a specific value (like 7) occur in array?
  - What is the maximum value in array?
  - What is the minimum value in array?



SEARCHING ARRAYS

- **Linear search is the most basic algorithm for searching**
  - Start at beginning of array (index 0)
  - Look at each element of array one at a time
  - Check if we have found what we are looking for
  - Stop at end of the array (index N-1)
  - This process is typically implemented with a loop

SEARCHING ARRAYS

```
// Linear searching for special value
float Special = 42;
for (int Pos = 0; Pos < SIZE; Pos++)
{
   if (Value[Pos] == Special)
      cout << "found value " << Special
           << " at position " << Pos << endl;
}
```

Loop over all array locations

Check for desired value in array

SEARCHING ARRAYS

```
// Linear searching for max and min values
float Minimum = Value[0];
float Maximum = Value[0];
for (int Pos = 1; Pos < SIZE; Pos++)
{
   if (Value[Pos] < Minimum)
      Minimum = Value[Pos];
   if (Value[Pos] > Maximum)
      Maximum = Value[Pos];
}
```

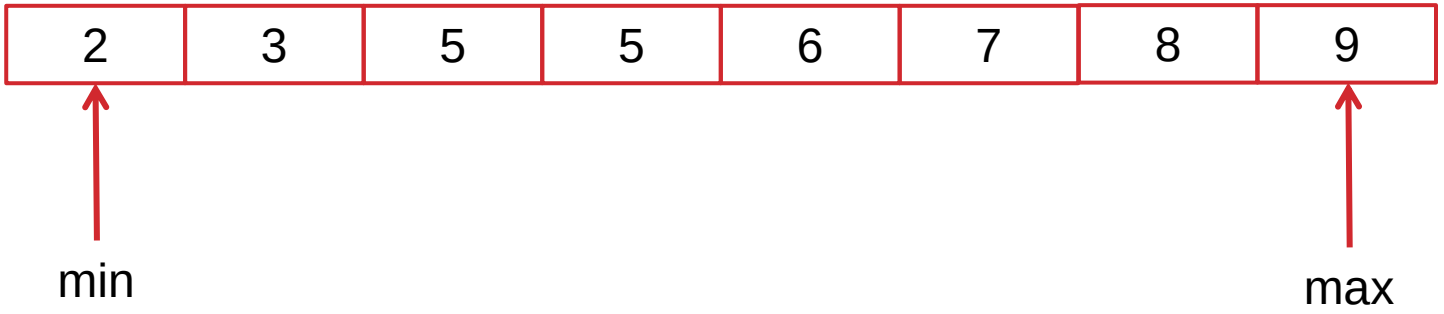
Initialize our best guess of min/max

Loop over all array locations

Update values of min/max as needed

BINARY SEARCH

- **What happens if we are given an array with sorted values?**
  - Now we know exactly where min/max should be
  - Minimum value always at location 0
  - Maximum value always at location N-1



BINARY SEARCH

- **The binary search algorithm can be used to search a sorted array for a specific value**
  - Look at middle element of sorted array
  - If equal to desired value, you found it
  - If less than desired value, search right half of array
  - If greater than desired value, search left half of array
  - Repeat until data is found (or no data left to search)

BINARY SEARCH

- **Search for value 7 in sorted array below**

- **Look at middle location (0+7)/2 = 3, which contains 5**

- **5 < 7, so search to right**

- **This cuts size of array we are searching in half**

BINARY SEARCH

- **Search for value 7 in unsearched array below**

- **Look at middle location (4+7)/2 = 5, which contains 7**

- **We found the desired value in only 2 searching steps!**

BINARY SEARCH

- **Search for value 2 in sorted array below**

- **Look at middle location (0+7)/2 = 3, which contains 5**

- **5 > 2, so search to left**

- **This cuts size of array we are searching in half**

BINARY SEARCH

- **Search for value 2 in unsearched array below**

- **Look at middle location (0+2)/2 = 1, which contains 3**

- **3 > 2, so search to left**

- **Now there is only one location to search!**

BINARY SEARCH

- **Search for value 2 in unsearched array below**

- **Look at middle location (0+0)/2 = 0, which contains 2**

- **We found the desired value in only 3 searching steps!**

BINARY SEARCH

- **This divide and conquer approach is very fast since the array we are searching is cut in half at each step**
  - Consider an array with 1024 sorted values
  - Searching we go from 1024 → 512 → 256 → 128 → 64 → 32 → 16 → 8 → 4 → 2 → 1
  - Only 10 steps needed to search array of 1024 elements

- **In general, binary search takes $\log_2 N$ steps to search a sorted array of N elements**
  - About 20 steps to search array of 1,000,000 elements
  - About 30 steps to search array of 1,000,000,000 elements

BINARY SEARCH

- **To implement binary search we need to:**
  - Keep track of the portion of the array we are searching
  - Min = smallest array index of unsearched portion
  - Max = largest array index of unsearched portion
  - Mid = (Min + Max) / 2 is middle position
  - We need to initialize Min=0 and Max=N-1
  - We need to update these values as we search

- **This can be implemented using iteration or using recursion**

BINARY SEARCH

```
// Iterative binary search
int Search(int Desired, int Data[], int Min, int Max)
{
  // Search array using divide and conquer approach
  int Mid = (Min + Max) / 2;
  while ((Data[Mid] != Desired) && (Max >= Min))
  {
    // Change min to search right half
    if (Data[Mid] < Desired)
      Min = Mid+1;
...
```

This loop will end when data is found or no locations are left to search

We change lower array index here to be 1 to right of midpoint

BINARY SEARCH

```
…
    // Change max to search left half
    else if (Data[Mid] > Desired)
       Max = Mid-1;

    // Update mid location
    Mid = (Min + Max) / 2;
  }
…
```

We change upper array index here to be 1 to left of midpoint

BINARY SEARCH

```
...
   // Return results of search
   if (Data[Mid] == Desired)
      return(Mid);
   else
      return(-1);
}
```

This returns the array index of desired data value or -1 if not found

BINARY SEARCH

```
// Recursive binary search
int Search( int Desired, int Data[], int Min, int Max )
{
  // Terminating conditions
  int Mid = (Min + Max ) / 2;
  if (Max < Min)
    return(-1);
  else if (Data[Mid] == Desired)
    return(Mid);
...
```

This returns the array index of desired data value or -1 if not found

BINARY SEARCH

```
...
  // Recursive call to search right half
  else if (Data[Mid] < Desired)
    return( Search( Desired, Data, Mid+1, Max ) );

  // Recursive call to search left half
  else if (Data[Mid] > Desired)
    return( Search( Desired, Data, Min, Mid-1 ) );
}
```

Notice how we search a smaller part of the array with each recursive function call

ARRAY SORTING

- **The basic idea of array sorting is to move data values in the array so they are in numerical or alphabetical order**

- **There are many applications that need sorted arrays**
  - Output array in ascending or descending order
  - Search array more efficiently using binary search

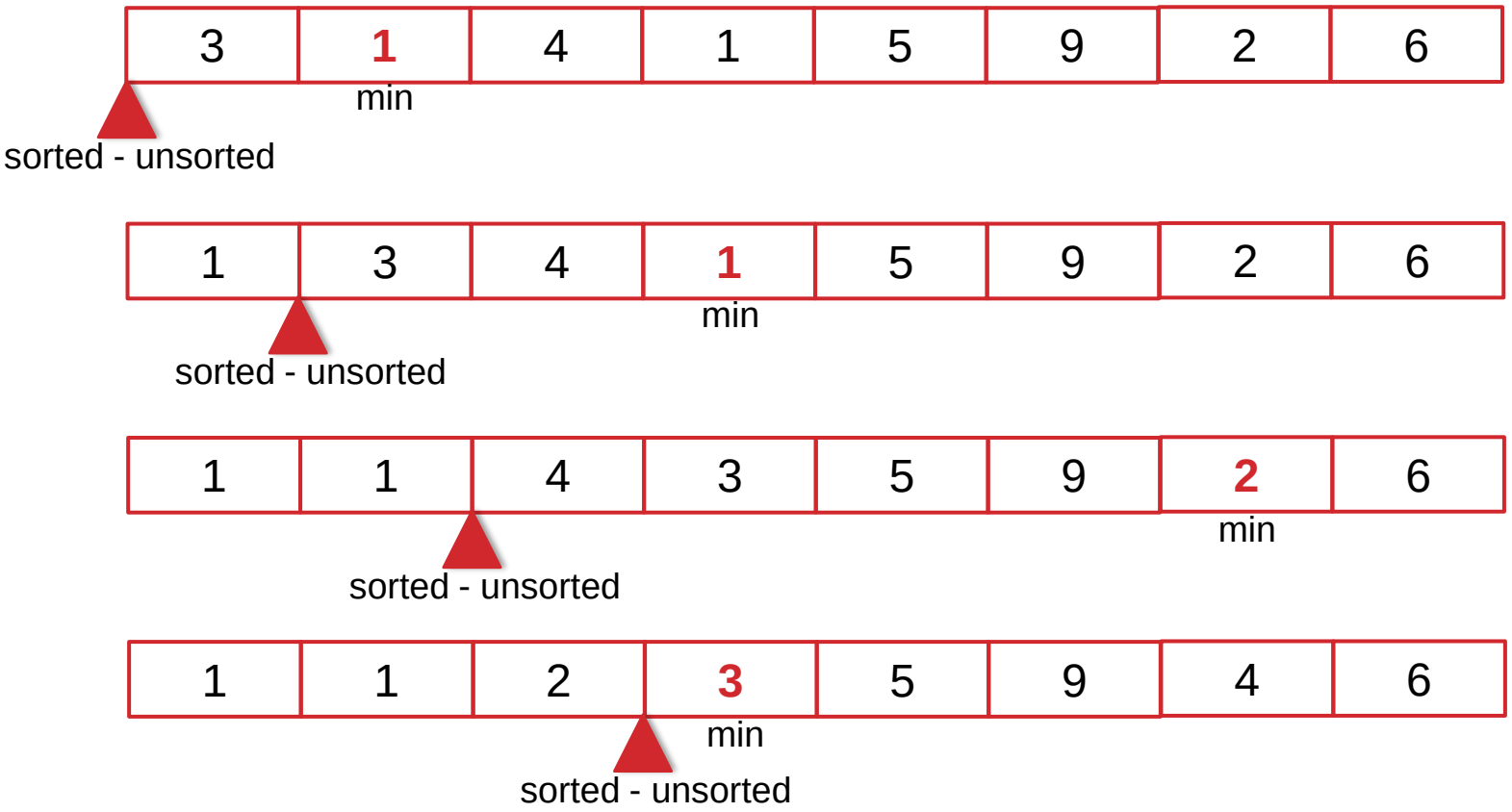
- **There are many algorithms for sorting arrays**
  - Some are easy to implement, others more complex
  - Some have fast run times, others are slower

ARRAY SORTING

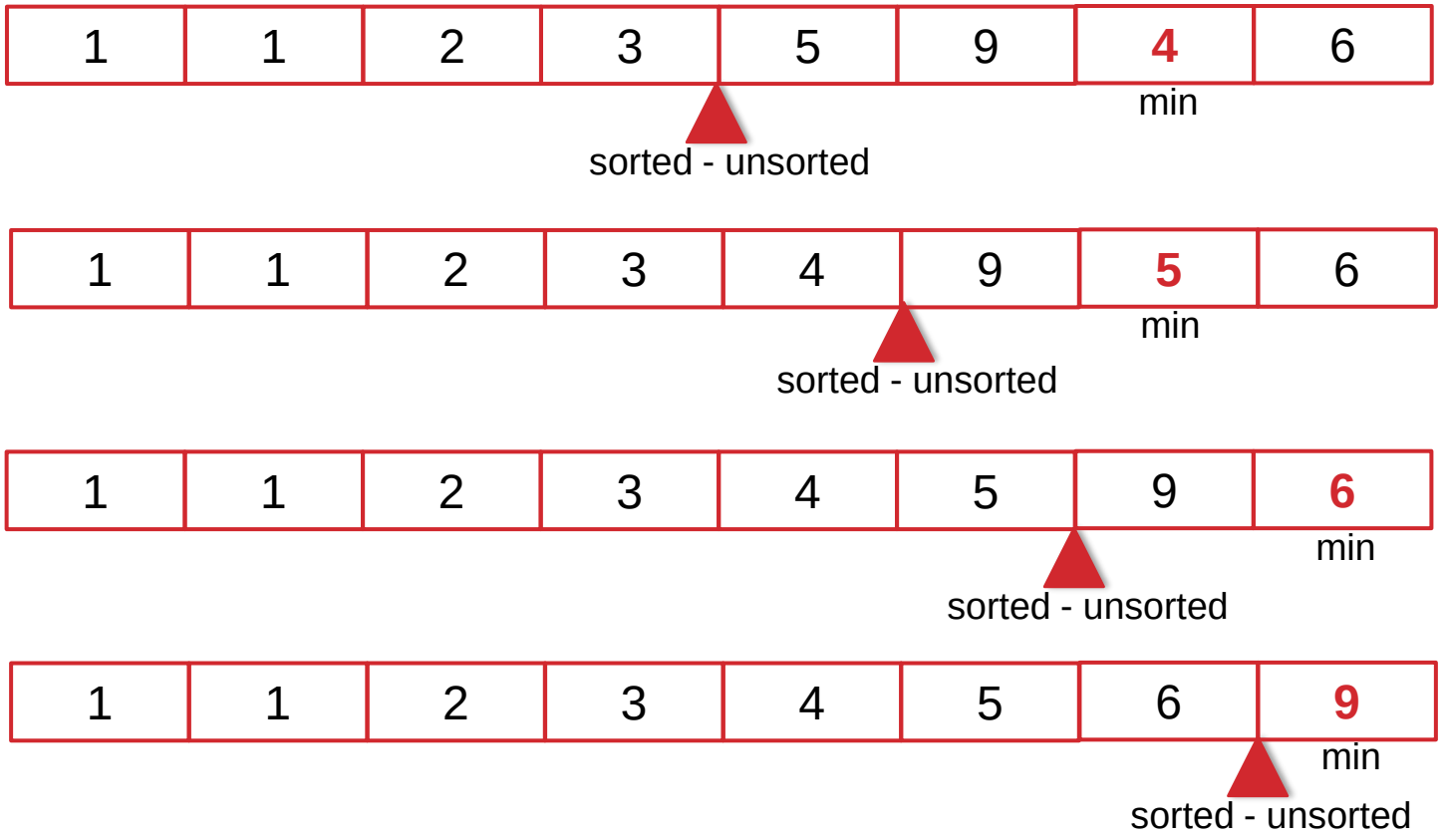
- **One easy algorithm to implement is selection sort**
  - Divide the array into two parts: sorted and unsorted
  - Find smallest value in the unsorted part of array
  - Swap value into end of sorted part of array
  - Repeat this process until the whole array is sorted

- **Consider an array containing the first 8 digits of PI**
  - Lets see what happens if we use selection sort
  - We show the array contents after each data swap

ARRAY SORTING



ARRAY SORTING



ARRAY SORTING

- **The array is sorted when the unsorted part is empty!**
- **In general, selection sort will take N "find minimum and swap iterations" to sort an array of N elements**
- **Each "find minimum value" step looks at N data values, so selection sort takes N^2 operations**
- **The implementation of selection sort is shown below**

ARRAY SORTING

```
// Initialize data to sort
const int SIZE = 10;
int Data[SIZE] = {3,1,4,1,5,9,2,6,5,3};

// Print unsorted data
for (int Index = 0; Index < SIZE; Index++)
   cout << Index << " " << Data[Index] << endl;
```

ARRAY SORTING

```
// Perform selection sort algorithm
for (int Index = 0; Index < SIZE; Index++)
{
   // Find smallest value in unsorted part
   int SmallPos = Index;
   for (int Pos = Index; Pos < SIZE; Pos++)
      if (Data[Pos] < Data[SmallPos])
         SmallPos = Pos;
   …
```

This loop executes N times moving the sorted-unsorted line

This loop executes N times to find the smallest data value

Notice that we start this loop at the beginning of the unsorted part of array

ARRAY SORTING

```
   …
   // Swap smallest value into sorted part
   int SmallVal = Data[SmallPos];
   Data[SmallPos] = Data[Index];
   Data[Index] = SmallVal;
}

// Print sorted data
for (int Index = 0; Index < SIZE; Index++)
   cout << Index << " " << Data[Index] << endl;
```

CALCULATING MEDIAN VALUE

- **The median is defined to be midpoint of set of values**
  - Half of the data values are larger
  - Half of the data values are smaller

- **Algorithm for calculating the median value**
  - Sort data into numerical order
  - Calculate midpoint = array_size / 2
  - Median value = data[midpoint]

- **Calculating the median is more work than finding the average, but it is considered to be a more robust statistic**

SUMMARY

- In this section, we described how linear search can be used to find the min/max or special values in an array
- Then, we described how “binary search” can be used to very quickly search for values in a sorted array
- Next, we introduced the “selection sort” algorithm and showed how it can be used to sort data
- Finally, we saw how a sorted array can be used to calculate the median value