Fibonacci Heap
FM-Index
Hash Table
Trie
Splay
Top Tree
BST
Tree
Heap
queue
Soft Heap
Stack
Data
Binary Hea
Soft Heap
Weak Heap
Doubly
Stack
Trie
Beap
Rope
BIT
BST
array
Queue
Array
Trie
Structur
Binomial Heap
Trie
ডেটা স্ট্রাকচার শিখি বাং
Queue
Beap
Rope
Heap
Treap
Suffix Array
BST
Tree
Splay Tree
Soft Heap

4.1 Introduction

Arrays

- Structures of related data items
- Static entity (same size throughout program)

A few types

- Pointer-based arrays (C-like)
- Arrays as objects (C++)

4.2 Arrays

Array

- Consecutive group of memory locations
- Same name and type (**`int`**, **`char`**, etc.)

To refer to an element

- Specify array name and position number (index)
- Format: arrayname[position number]
- First element at position 0

N-element array c

`c[ 0 ]`, **`c[ 1 ]`** … **`c[ n - 1 ]`**

- Nth element as position N-1

4.2 Arrays

Array elements like other variables

- Assignment, printing for an integer array **`c`**

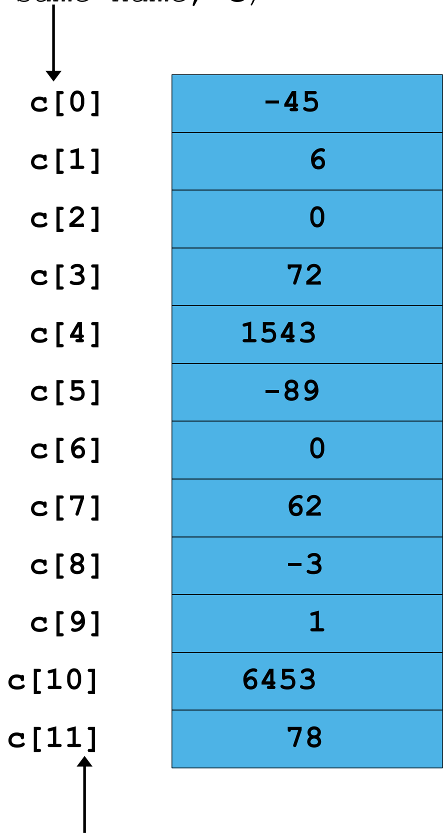
```
c[ 0 ] =  3;
cout << c[ 0 ];
```

Can perform operations inside subscript

`c[ 5 - 2 ]` same as **`c[3]`**

4.2 Arrays

Name of array (Note that all elements of this array have the same name, **c**)



c[0]	**-45**
c[1]	**6**
c[2]	**0**
c[3]	**72**
c[4]	**1543**
c[5]	**-89**
c[6]	**0**
c[7]	**62**
c[8]	**-3**
c[9]	**1**
c[10]	**6453**
c[11]	**78**

Position number of the element within array **c**

4.3 Declaring Arrays

When declaring arrays, specify

- Name
- Type of array
  - Any data type
- Number of elements
- *type arrayName*[*arraySize*];

```
int c[ 10 ];  // array of 10 integers
float d[ 3284 ]; // array of 3284 floats
```

Declaring multiple arrays of same type

- Use comma separated list, like regular variables

```
int b[ 100 ], x[ 27 ];
```

4.4 Examples Using Arrays

Initializing arrays

- For loop
  - Set each element
- Initializer list
  - Specify each element when array declared

  ```
  int n[ 5 ] = { 1, 2, 3, 4, 5 };
  ```

  - If not enough initializers, rightmost elements 0
  - If too many syntax error
- To set every element to same value

  ```
  int n[ 5 ] = { 0 };
  ```

- If array size omitted, initializers determine size

  ```
  int n[] = { 1, 2, 3, 4, 5 };
  ```

  - 5 initializers, therefore 5 element array

fig04_03.cpp (1 of 2)

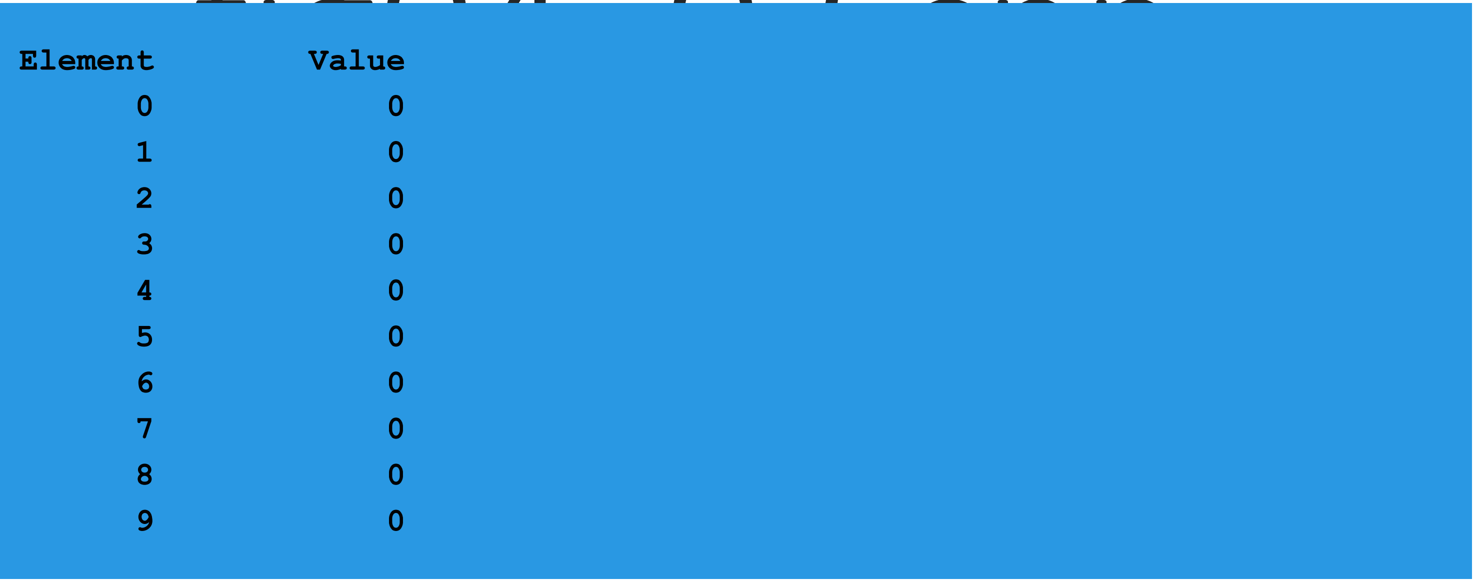
Declare a 10-element array of integers.

Initialize array to **`0`** using a for loop. Note that the array has elements **`n[0]`** to **`n[9]`**.

```
// FIG. 4.3: FIG04_03.CPP
// INITIALIZING AN ARRAY.
#INCLUDE <IOSTREAM>

USING STD::COUT;
USING STD::ENDL;

#INCLUDE <IOMANIP>

USING STD::SETW;
```

```
   RETURN 0; // INDICATES SUCCESSFUL TERMINATION

} // END MAIN
```

fig04_03.cpp



```
Element        Value
      0            0
      1            0
      2            0
      3            0
      4            0
      5            0
      6            0
      7            0
      8            0
      9            0
```

fig04_04.cpp (1 of 1)

Note the use of the initializer list.

```
// FIG. 4.4: FIG04_04.CPP
// INITIALIZING AN ARRAY WITH A DECLARATION.
#INCLUDE <IOSTREAM>

USING STD::COUT;
USING STD::ENDL;

#INCLUDE <IOMANIP>

USING STD::SETW;
```

ELEMENT	VALUE
0	32
1	27
2	64
3	18
4	95
5	14
6	90
7	70
8	60
9	37

4.4 Examples Using Arrays

Array size

- Can be specified with constant variable (**`const`**)
  - **`const int size = 20;`**
- Constants cannot be changed
- Constants must be initialized when declared
- Also called named constants or read-only variables

fig04_05.cpp (1 of 2)

Note use of **`const`** keyword. Only **`const`** variables can specify array sizes.

The program becomes more scalable when we set the array size using a **`const`** variable. We can change **`arraySize`**, and all the loops will still work (otherwise, we'd have to update every loop in the program).

```
// FIG. 4.5: FIG04_05.CPP
// INITIALIZE ARRAY S TO THE EVEN INTEGERS FROM 2 TO 2
#INCLUDE <IOSTREAM>

USING STD::COUT;
USING STD::ENDL;

#INCLUDE <IOMANIP>

USING STD::SETW;
```

fig04_05.cpp

```
   // OUTPUT CONTENTS OF ARRAY S IN TABULAR FORMAT
   FOR ( INT J = 0; J < ARRAYSIZE; J++ )
      COUT << SETW( 7 ) << J << SETW( 13 ) << S[ J ] << ENDL;

   RETURN 0; // INDICATES SUCCESSFUL TERMINATION

} // END MAIN
```

```
Element        Value
      0            2
      1            4
      2            6
      3            8
      4           10
      5           12
      6           14
      7           16
      8           18
      9           20
```

fig04_06.cpp output (1 of 1)

```
// FIG. 4.6: FIG04_06.CPP
// USING A PROPERLY INITIALIZED CONSTANT VARIABLE.
#INCLUDE <IOSTREAM>

USING STD::COUT;
USING STD::ENDL;

INT MAIN()
{
  CONST INT X = 7; // INITIALIZED CONSTANT VARIABLE

  COUT << "THE VALUE OF CONSTANT VARIABLE X IS: "
       << X << ENDL;

  RETURN 0; // INDICATES SUCCESSFUL TERMINATION

} // END MAIN
```

Proper initialization of **`const`** variable.

```
The value of constant variable x is: 7
```

fig04_ output (1 of 1)

```
// FIG. 4.7: FIG04_07.CPP
// A CONST OBJECT MUST BE INITIALIZED.

INT MAIN()
{
   CONST INT X; // ERROR: X MUST BE INITIALIZED

   X = 7;       // ERROR: CANNOT MODIFY A CONST VARIABLE

  RETURN 0;    // INDICATES SUCCESSFUL TERMINATION

} // END MAIN
```

Uninitialized **const** results in a syntax error. Attempting to modify the **const** is another error.

```
d:\cpphtp4_examples\ch04\Fig04_07.cpp(6) : error C2734: 'x' :
   const object must be initialized if not extern
d:\cpphtp4_examples\ch04\Fig04_07.cpp(8) : error C2166:
   l-value specifies const object
```

fig04_08.cpp output (1 of 1)

```
Total of array element values is 55
```

```
// FIG. 4.8: FIG04_08.CPP
// COMPUTE THE SUM OF THE ELEMENTS OF THE ARRAY.
#INCLUDE <IOSTREAM>

USING STD::COUT;
```

```
INT MAIN()
{
   CONST INT ARRAYSIZE = 10;
```

fig04_09.cpp (1 of 2)

```
// FIG. 4.9: FIG04_09.CPP
// HISTOGRAM PRINTING PROGRAM.
#INCLUDE <IOSTREAM>

USING STD::COUT;
USING STD::ENDL;

#INCLUDE <IOMANIP>

USING STD::SETW;

INT MAIN()
{
  CONST INT ARRAYSIZE = 10;
  INT N[ ARRAYSIZE ] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };

  COUT << "ELEMENT" << SETW( 13 ) << "VALUE"
       << SETW( 17 ) << "HISTOGRAM" << ENDL;

  // FOR EACH ELEMENT OF ARRAY N, OUTPUT A BAR IN HISTOGRAM
  FOR ( INT I = 0; I < ARRAYSIZE; I++ ) {
    COUT << SETW( 7 ) << I << SETW( 13 )
         << N[ I ] << SETW( 9 );

    FOR ( INT J = 0; J < N[ I ]; J++ )   // PRINT ONE BAR
      COUT << '*';
```

Prints asterisks corresponding to size of array element, **`n[i]`**.

fig04_09.cpp output (1 of 1)

```
         COUT << ENDL; // START NEXT LINE OF OUTPUT

      } // END OUTER FOR STRUCTURE

      RETURN 0; // INDICATES SUCCESSFUL TERMINATION

   } // END MAIN
```

```
Element        Value        Histogram
      0           19        *******************
      1            3        ***
      2           15        ***************
      3            7        *******
      4           11        ***********
      5            9        *********
      6           13        *************
      7            5        *****
      8           17        *****************
      9            1        *
```

fig04_10.cpp (1 of 2)

Remake of old program to roll dice. An array is used instead of 6 regular variables, and the proper element can be updated easily (without needing a **`switch`**).

This creates a number between 1 and 6, which determines the index of **`frequency[]`** that should be incremented.

```
// FIG. 4.10: FIG04_10.CPP
// ROLL A SIX-SIDED DIE 6000 TIMES.
#INCLUDE <IOSTREAM>

USING STD::COUT;
USING STD::ENDL;

#INCLUDE <IOMANIP>

USING STD::SETW;
```

fig04_10.cpp output (1 of 1)

```

   COUT << "FACE" << SETW( 13 ) << "FREQUENCY" << ENDL;

   // OUTPUT FREQUENCY ELEMENTS 1-6 IN TABULAR FORMAT
   FOR ( INT FACE = 1; FACE < ARRAYSIZE; FACE++ )
      COUT << SETW( 4 ) << FACE
           << SETW( 13 ) << FREQUENCY[ FACE ] << ENDL;

   RETURN 0; // INDICATES SUCCESSFUL TERMINATION

} // END MAIN
```

```
Face    Frequency
   1         1003
   2         1004
   3          999
   4          980
   5         1013
   6         1001
```

fig04_11.cpp (1 of 2)

```
// FIG. 4.11: FIG04_11.CPP
// STUDENT POLL PROGRAM.
#INCLUDE <IOSTREAM>

USING STD::COUT;
USING STD::ENDL;

#INCLUDE <IOMANIP>

USING STD::SETW;
```

fig04_11.cpp (2 of 2)

```
   // FOR EACH ANSWER, SELECT VALUE OF AN ELEMENT OF ARRAY
   // RESPONSES AND USE THAT VALUE AS SUBSCRIPT IN ARRAY
   // FREQUENCY TO DETERMINE ELEMENT TO INCREMENT
   FOR ( INT ANSWER = 0; ANSWER < RESPONSESIZE; ANSWER++ )
      ++FREQUENCY[ RESPONSES[ANSWER] ];

   // DISPLAY RESULTS
   COUT << "RATING" << SETW( 17 ) << "FREQUENCY" << ENDL;

   // OUTPUT FREQUENCIES IN TABULAR FORMAT
   FOR ( INT RATING = 1; RATING < FREQUENCYSIZE; RATING++ )
      COUT << SETW( 6 ) << RATING
           << SETW( 17 ) << FREQUENCY[ RATING ] << ENDL;

   RETURN 0; // INDICATES SUCCESSFUL TERMINATION

} // END MAIN
```

`responses[answer]` is the rating (from 1 to 10). This determines the index in **`frequency[]`** to increment.

RATING	FREQUENCY
1	2
2	2
3	2
4	2
5	5
6	11
7	5
8	7
9	1
10	3

4.4 Examples Using Arrays

Strings (more in ch. 5)

- Arrays of characters
- All strings end with **`null`** (`'\0'`)
- Examples
  - `char string1[] = "hello";`
    - **`Null`** character implicitly added
    - **`string1`** has 6 elements
  - `char string1[] = { 'h', 'e', 'l', 'l',    'o', '\0’ };`
- Subscripting is the same

  `String1[ 0 ]` is `'h'`

  `string1[ 2 ]` is `'l'`

4.4 Examples Using Arrays

Input from keyboard

```
char string2[ 10 ];
cin >> string2;
```

- Puts user input in string
  - Stops at first whitespace character
  - Adds **`null`** character
- If too much text entered, data written beyond array
  - We want to avoid this (section 5.12 explains how)

Printing strings

- **`cout << string2 << endl;`**
  - Does not work for other array types
- Characters printed until **`null`** found

fig04_12.cpp (1 of 2)

```
// FIG. 4_12: FIG04_12.CPP
// TREATING CHARACTER ARRAYS AS STRINGS.
#INCLUDE <IOSTREAM>

USING STD::COUT;
USING STD::CIN;
USING STD::ENDL;

INT MAIN()
{
  CHAR STRING1[ 20 ],              // RESERVES 20 CHARACTERS
  CHAR STRING2[] = "STRING LITERAL"; // RESERVES 15 CH

  // READ STRING FROM USER INTO ARRAY STRING2
  COUT << "ENTER THE STRING \"HELLO THERE\": ";
  CIN >> STRING1;  // READS "HELLO" [SPACE TERMINATES INPUT]

  // OUTPUT STRINGS
  COUT << "STRING1 IS: " << STRING1
       << "\NSTRING2 IS: " << STRING2;

  COUT << "\NSTRING1 WITH SPACES BETWEEN CHARACTERS IS:\N";

```

Two different ways to declare strings. **string2** is initialized, and its size determined automatically .

Examples of reading strings from the keyboard and printing them out.

```
    // OUTPUT CHARACTERS UNTIL NULL CHARACTER IS REACHED
    FOR ( INT I = 0; STRING1[ I ] != '\0'; I++ )
       COUT << STRING1[ I ] << ' ';

    CIN >> STRING1; // READS "THERE"
    COUT << "\NSTRING1 IS: " << STRING1 << ENDL;

    RETURN 0; // INDICATES SUCCESSFUL TERMINATION

 } // END MAIN
```

fig04_12.cpp output (1 of 1)

Can access the characters in a string using array notation. The loop ends when the **`null`** character is found.

```
Enter the string "hello there": hello there
string1 is: hello
string2 is: string literal
string1 with spaces between characters is:
h e l l o
string1 is: there
```

4.4 Examples Using Arrays

Recall static storage (chapter 3)

- If **`static`**, local variables save values between function calls
- Visible only in function body
- Can declare local arrays to be static
  - Initialized to zero

```
static int array[3];
```

If not static

- Created (and destroyed) in every function call

fig04_13.cpp (1 of 3)

```
// FIG. 4.13: FIG04_13.CPP
// STATIC ARRAYS ARE INITIALIZED TO ZERO.
#INCLUDE <IOSTREAM>

USING STD::COUT;
USING STD::ENDL;

VOID STATICARRAYINIT( VOID );      // FUNCTION PROTOTYPE
VOID AUTOMATICARRAYINIT( VOID );   // FUNCTION PROTOTYPE

```

fig04_13.cpp (2 of 3)

```
// FUNCTION TO DEMONSTRATE A STATIC LOCAL ARRAY
VOID STATICARRAYINIT( VOID )
{
   // INITIALIZES ELEMENTS TO 0 FIRST TIME FUNCTION IS
   STATIC INT ARRAY1[ 3 ];

   COUT << "\NVALUES ON ENTERING STATICARRAYINIT:\N";

   // OUTPUT CONTENTS OF ARRAY1
   FOR ( INT I = 0; I < 3; I++ )
      COUT << "ARRAY1[" << I << "] = " << ARRAY1[ I ] << "  ";

   COUT << "\NVALUES ON EXITING STATICARRAYINIT:\N";

   // MODIFY AND OUTPUT CONTENTS OF ARRAY1
   FOR ( INT J = 0; J < 3; J++ )
      COUT << "ARRAY1[" << J << "] = "
           << ( ARRAY1[ J ] += 5 ) << "  ";

} // END FUNCTION STATICARRAYINIT

```

Static array, initialized to zero on first function call.

Array data is changed; the modified values stay.

fig04_13.cpp (3 of 3)

```
// FUNCTION TO DEMONSTRATE AN AUTOMATIC LOCAL ARRAY
VOID AUTOMATICARRAYINIT( VOID )
{
   // INITIALIZES ELEMENTS EACH TIME FUNCTION IS CALLED
   INT ARRAY2[ 3 ] = { 1, 2, 3 };

   COUT << "\N\NVALUES ON ENTERING AUTOMATICARRAYINIT:\N";

   // OUTPUT CONTENTS OF ARRAY2
   FOR ( INT I = 0; I < 3; I++ )
      COUT << "ARRAY2[" << I << "] = " << ARRAY2[ I ] << "  ";

   COUT << "\NVALUES ON EXITING AUTOMATICARRAYINIT:\N";

   // MODIFY AND OUTPUT CONTENTS OF ARRAY2
   FOR ( INT J = 0; J < 3; J++ )
      COUT << "ARRAY2[" << J << "] = "
           << ( ARRAY2[ J ] += 5 ) << "  ";

} // END FUNCTION AUTOMATICARRAYINIT
```

Automatic array, recreated with every function call.

Although the array is changed, it will be destroyed when the function exits and the changes will be lost.

```
FIRST CALL TO EACH FUNCTION:


VALUES ON ENTERING STATICARRAYINIT:

ARRAY1[0] = 0  ARRAY1[1] = 0  ARRAY1[2] = 0

VALUES ON EXITING STATICARRAYINIT:

ARRAY1[0] = 5  ARRAY1[1] = 5  ARRAY1[2] = 5


VALUES ON ENTERING AUTOMATICARRAYINIT:

ARRAY2[0] = 1  ARRAY2[1] = 2  ARRAY2[2] = 3

VALUES ON EXITING AUTOMATICARRAYINIT:

ARRAY2[0] = 6  ARRAY2[1] = 7  ARRAY2[2] = 8


SECOND CALL TO EACH FUNCTION:


VALUES ON ENTERING STATICARRAYINIT:

ARRAY1[0] = 5  ARRAY1[1] = 5  ARRAY1[2] = 5

VALUES ON EXITING STATICARRAYINIT:

ARRAY1[0] = 10  ARRAY1[1] = 10  ARRAY1[2] = 10


VALUES ON ENTERING AUTOMATICARRAYINIT:

ARRAY2[0] = 1  ARRAY2[1] = 2  ARRAY2[2] = 3

VALUES ON EXITING AUTOMATICARRAYINIT:

ARRAY2[0] = 6  ARRAY2[1] = 7  ARRAY2[2] = 8
```

4.5 Passing Arrays to Functions

Specify name without brackets

- To pass array **`myArray`** to **`myFunction`**

```
int myArray[ 24 ];
myFunction( myArray, 24 );
```

- Array size usually passed, but not required
  - Useful to iterate over all elements

4.5 Passing Arrays to Functions

Arrays passed-by-reference

- Functions can modify original array data
- Value of name of array is address of first element
  - Function knows where the array is stored
  - Can change original memory locations

Individual array elements passed-by-value

- Like regular variables
- `square( myArray[3] );`

4.5 Passing Arrays to Functions

Functions taking arrays

- Function prototype
  - `void modifyArray( int b[], int arraySize );`
  - `void modifyArray( int [], int );`
    - Names optional in prototype
  - Both take an integer array and a single integer
- No need for array size between brackets
  - Ignored by compiler
- If declare array parameter as `const`
  - Cannot be modified (compiler error)
  - `void doNotModify( const int [] );`

fig04_14.cpp (1 of 3)

Syntax for accepting an array in parameter list.

```
// FIG. 4.14: FIG04_14.CPP
// PASSING ARRAYS AND INDIVIDUAL ARRAY ELEMENTS TO FUNCTIONS.
#INCLUDE <IOSTREAM>

USING STD::COUT;
USING STD::ENDL;

#INCLUDE <IOMANIP>

USING STD::SETW;
```

fig04_14.cpp (2 of 3)

```

   COUT << ENDL;

   // PASS ARRAY A TO MODIFYARRAY BY REFERENCE
   MODIFYARRAY( A, ARRAYSIZE );

   COUT << "THE VALUES OF THE MODIFIED ARRAY ARE:\N";

   // OUTPUT MODIFIED ARRAY
   FOR ( INT J = 0; J < ARRAYSIZE; J++ )
      COUT << SETW( 3 ) << A[ J ];

   // OUTPUT VALUE OF A[ 3 ]
   COUT << "\N\N\N"
        << "EFFECTS OF PASSING ARRAY ELEMENT BY VALUE:"
        << "\N\NTHE VALUE OF A[3] IS " << A[ 3 ] << '\N';

   // PASS ARRAY ELEMENT A[ 3 ] BY VALUE
   MODIFYELEMENT( A[ 3 ] );

   // OUTPUT VALUE OF A[ 3 ]
   COUT << "THE VALUE OF A[3] IS " << A[ 3 ] << ENDL;

   RETURN 0; // INDICATES SUCCESSFUL TERMINATION

} // END MAIN
```

Pass array name (**a**) and size to function. Arrays are passed-by-reference.

Pass a single array element by value; the original cannot be modified.

fig04_14.cpp (3 of 3)

```

// IN FUNCTION MODIFYARRAY, "B" POINTS TO
// THE ORIGINAL ARRAY "A" IN MEMORY
VOID MODIFYARRAY( INT B[], INT SIZEOFARRAY )
{
   // MULTIPLY EACH ARRAY ELEMENT BY 2
   FOR ( INT K = 0; K < SIZEOFARRAY; K++ )
      B[ K ] *= 2;

} // END FUNCTION MODIFYARRAY

// IN FUNCTION MODIFYELEMENT, "E" IS A LOCAL COPY OF
// ARRAY ELEMENT A[ 3 ] PASSED FROM MAIN
VOID MODIFYELEMENT( INT E )
{
   // MULTIPLY PARAMETER BY 2
   COUT << "VALUE IN MODIFYELEMENT IS "
        << ( E *= 2 ) << ENDL;

} // END FUNCTION MODIFYELEMENT
```

Although named **b**, the array points to the original array **a**. It can modify **a**’s data.

Individual array elements are passed by value, and the originals cannot be changed.

```
EFFECTS OF PASSING ENTIRE ARRAY BY REFERENCE:


THE VALUES OF THE ORIGINAL ARRAY ARE:

  0  1  2  3  4

THE VALUES OF THE MODIFIED ARRAY ARE:

  0  2  4  6  8




EFFECTS OF PASSING ARRAY ELEMENT BY VALUE:


THE VALUE OF A[3] IS 6

VALUE IN MODIFYELEMENT IS 12

THE VALUE OF A[3] IS 6
```

fig04_15.cpp (1 of 2)

```
// FIG. 4.15: FIG04_15.CPP
// DEMONSTRATING THE CONST TYPE QUALIFIER.
#INCLUDE <IOSTREAM>

USING STD::COUT;
USING STD::ENDL;

VOID TRYTOMODIFYARRAY( CONST INT [] ); // FUNCTION PROTOTYPE

INT MAIN()
{
   INT A[] = { 10, 20, 30 };

   TRYTOMODIFYARRAY( A );

   COUT << A[ 0 ] << ' ' << A[ 1 ] << ' ' << A[ 2 ] << '\N';

   RETURN 0; // INDICATES SUCCESSFUL TERMINATION

} // END MAIN

```

Array parameter declared as **const**. Array cannot be modified, even though it is passed by reference.

fig04_15.cpp output (1 of 1)

```
// IN FUNCTION TRYTOMODIFYARRAY, "B" CANNOT BE USED
// TO MODIFY THE ORIGINAL ARRAY "A" IN MAIN.
VOID TRYTOMODIFYARRAY( CONST INT B[] )
{
   B[ 0 ] /= 2;    // ERROR
   B[ 1 ] /= 2;    // ERROR
   B[ 2 ] /= 2;    // ERROR

} // END FUNCTION TRYTOMODIFYARRAY
```

```
d:\cpphtp4_examples\ch04\Fig04_15.cpp(26) : error C2166:
   l-value specifies const object
d:\cpphtp4_examples\ch04\Fig04_15.cpp(27) : error C2166:
   l-value specifies const object
d:\cpphtp4_examples\ch04\Fig04_15.cpp(28) : error C2166:
   l-value specifies const object
```

4.6 Sorting Arrays

Sorting data

- Important computing application
- Virtually every organization must sort some data
  - Massive amounts must be sorted

Bubble sort (sinking sort)

- Several passes through the array
- Successive pairs of elements are compared
  - If increasing order (or identical), no change
  - If decreasing order, elements exchanged
- Repeat these steps for every element

4.6 Sorting Arrays

Example:

- Go left to right, and exchange elements as necessary
  - One pass for each element
- Original: 3 4 2 7 6
- Pass 1: 3 2 4 6 7 (elements exchanged)
- Pass 2: 2 3 4 6 7
- Pass 3: 2 3 4 6 7 (no changes needed)
- Pass 4: 2 3 4 6 7
- Pass 5: 2 3 4 6 7
- Small elements "bubble" to the top (like 2 in this example)

4.6 Sorting Arrays

Swapping variables

```
int x = 3, y = 4;
y = x;
x = y;
```

What happened?

- Both x and y are 3!
- Need a temporary variable

Solution

```
int x = 3, y = 4, temp = 0;
temp = x;  // temp gets 3
x = y;     // x gets 4
y = temp;  // y gets 3
```

fig04_16.cpp
(1 of 3)

```
// FIG. 4.16: FIG04_16.CPP
// THIS PROGRAM SORTS AN ARRAY'S VALUES INTO ASCENDING ORDER.
#INCLUDE <IOSTREAM>

USING STD::COUT;
USING STD::ENDL;

#INCLUDE <IOMANIP>

USING STD::SETW;

INT MAIN()
{
   CONST INT ARRAYSIZE = 10;  // SIZE OF ARRAY A
   INT A[ ARRAYSIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
   INT HOLD;  // TEMPORARY LOCATION USED TO SWAP ARRAY ELEMENTS

   COUT << "DATA ITEMS IN ORIGINAL ORDER\N";

   // OUTPUT ORIGINAL ARRAY
   FOR ( INT I = 0; I < ARRAYSIZE; I++ )
      COUT << SETW( 4 ) << A[ I ];

```

fig04_16.cpp (2 of 3)

```
// BUBBLE SORT
// LOOP TO CONTROL NUMBER OF PASSES
FOR ( INT PASS = 0; PASS < ARRAYSIZE - 1; PASS++ )

   // LOOP TO CONTROL NUMBER OF COMPARISONS PER PASS
   FOR ( INT J = 0; J < ARRAYSIZE - 1; J++ )

      // COMPARE SIDE-BY-SIDE ELEMENTS AND SWAP THEM IF
      // FIRST ELEMENT IS GREATER THAN SECOND ELEMENT
      IF ( A[ J ] > A[ J + 1 ] ) {
         HOLD = A[ J ];
         A[ J ] = A[ J + 1 ];
         A[ J + 1 ] = HOLD;

      } // END IF

```

Do a pass for each element in the array.

If the element on the left (index `j`) is larger than the element on the right (index `j + 1`), then we swap them. Remember the need of a temp variable.

fig04_16.cpp output (1 of 1)

```
    COUT << "\NDATA ITEMS IN ASCENDING ORDER\N";

    // OUTPUT SORTED ARRAY
    FOR ( INT K = 0; K < ARRAYSIZE; K++ )
       COUT << SETW( 4 ) << A[ K ];

    COUT << ENDL;

    RETURN 0;  // INDICATES SUCCESSFUL TERMINATION

 } // END MAIN
```

```
Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

4.7 Case Study: Computing Mean, Median and Mode Using Arrays

Mean

- Average (sum/number of elements)

Median

- Number in middle of sorted list
- 1, 2, 3, 4, 5 (3 is median)
- If even number of elements, take average of middle two

Mode

- Number that occurs most often
- 1, 1, 1, 2, 3, 3, 4, 5 (1 is mode)

fig04_17.cpp (1 of 8)

```
// FIG. 4.17: FIG04_17.CPP
// THIS PROGRAM INTRODUCES THE TOPIC OF SURVEY DATA ANALYSIS.
// IT COMPUTES THE MEAN, MEDIAN, AND MODE OF THE DATA.
#INCLUDE <IOSTREAM>

USING STD::COUT;
USING STD::ENDL;
USING STD::FIXED;
USING STD::SHOWPOINT;

```

fig04_17.cpp (2 of 8)

```
    INT FREQUENCY[ 10 ] = { 0 }; // INITIALIZE ARRAY FREQUENCY

    // INITIALIZE ARRAY RESPONSES
    INT RESPONSE[ RESPONSESIZE ] =
        { 6, 7, 8, 9, 8, 7, 8, 9, 8, 9,
          7, 8, 9, 5, 9, 8, 7, 8, 7, 8,
          6, 7, 8, 9, 3, 9, 8, 7, 8, 7,
          7, 8, 9, 8, 9, 8, 9, 7, 8, 9,
          6, 7, 8, 7, 8, 7, 9, 8, 9, 2,
          7, 8, 9, 8, 9, 8, 9, 7, 5, 3,
```

fig04_17.cpp (3 of 8)

We cast to a double to get decimal points for the average (instead of an integer).

```
// CALCULATE AVERAGE OF ALL RESPONSE VALUES
VOID MEAN( CONST INT ANSWER[], INT ARRAYSIZE )
{
   INT TOTAL = 0;

   COUT << "********\N  MEAN\N********\N";

   // TOTAL RESPONSE VALUES
   FOR ( INT I = 0; I < ARRAYSIZE; I++ )
      TOTAL += ANSWER[ I ];
```

fig04_17.cpp (4 of 8)

```
// SORT ARRAY AND DETERMINE MEDIAN ELEMENT'S VALUE
VOID MEDIAN( INT ANSWER[], INT SIZE )
{
   COUT << "\N********\N MEDIAN\N********\N"
      << "THE UNSORTED ARRAY OF RESP

   PRINTARRAY( ANSWER, SIZE );  // OUTP

   BUBBLESORT( ANSWER, SIZE );  // SORT ARRAY

   COUT << "\N\NTHE SORTED ARRAY IS";
   PRINTARRAY( ANSWER, SIZE );  // OUTPUT SORTED ARRAY

   // DISPLAY MEDIAN ELEMENT
   COUT << "\N\NTHE MEDIAN IS ELEMENT " << SIZE / 2
      << " OF\NTHE SORTED " << SIZE
      << " ELEMENT ARRAY.\NFOR THIS RUN THE MEDIAN IS "
      << ANSWER[ SIZE / 2 ] << "\N\N";

} // END FUNCTION MEDIAN

```

Sort array by passing it to a function. This keeps the program modular.

fig04_17.cpp (5 of 8)

```
// DETERMINE MOST FREQUENT RESPONSE
VOID MODE( INT FREQ[], INT ANSWER[], INT SIZE )
{
   INT LARGEST = 0;    // REPRESENTS LARGEST FREQUENCY
  INT MODEVALUE = 0;  // REPRESENTS MOST FREQUENT RESPONSE

  COUT << "\N********\N  MODE\N********\N";

  // INITIALIZE FREQUENCIES TO 0
  FOR ( INT I = 1; I <= 9; I++ )
    FREQ[ I ] = 0;

  // SUMMARIZE FREQUENCIES
  FOR ( INT J = 0; J < SIZE; J++ )
    ++FREQ[ ANSWER[ J ] ];

  // OUTPUT HEADERS FOR RESULT COLUMNS
  COUT << "RESPONSE" << SETW( 11 ) << "FREQUENCY"
       << SETW( 19 ) << "HISTOGRAM\N\N" << SETW( 55 )
       << "1    1    2    2\N" << SETW( 56 )
       << "5    0    5    0    5\N\N";

```

fig04_17.cpp (6 of 8)

```
// OUTPUT RESULTS
FOR ( INT RATING = 1; RATING <= 9; RATING++ ) {
  COUT << SETW( 8 ) << RATING << SETW( 11 )
     << FREQ[ RATING ] << "       ";

  // KEEP TRACK OF MODE VALUE AND LARGEST FREQUENCY VALUE
  IF ( FREQ[ RATING ] > LARGEST ) {
    LARGEST = FREQ[ RATING ];
    MODEVALUE = RATING;

  } // END IF

  // OUTPUT HISTOGRAM BAR REPRESENTING FREQUENCY VALUE
  FOR ( INT K = 1; K <= FREQ[ RATING ]; K++ )
    COUT << '*';

  COUT << '\N';  // BEGIN NEW LINE OF OUTPUT

} // END OUTER FOR

// DISPLAY THE MODE VALUE
COUT << "THE MODE IS THE MOST FREQUENT VALUE.\N"
   << "FOR THIS RUN THE MODE IS " << MODEVALUE
   << " WHICH OCCURRED " << LARGEST << " TIMES." << ENDL;
```

The mode is the value that occurs most often (has the highest value in **`freq`**).

fig04_17.cpp (7 of 8)

```

// FUNCTION THAT SORTS AN ARRAY WITH BUBBLE SORT ALGORITHM
VOID BUBBLESORT( INT A[], INT SIZE )
{
   INT HOLD;  // TEMPORARY LOCATION USED TO SWAP ELEMENTS

   // LOOP TO CONTROL NUMBER OF PASSES
   FOR ( INT PASS = 1; PASS < SIZE; PASS++ )

      // LOOP TO CONTROL NUMBER OF COMPARISONS PER PASS
      FOR ( INT J = 0; J < SIZE - 1; J++ )

         // SWAP ELEMENTS IF OUT OF ORDER
         IF ( A[ J ] > A[ J + 1 ] ) {
            HOLD = A[ J ];
            A[ J ] = A[ J + 1 ];
            A[ J + 1 ] = HOLD;

         } // END IF

} // END FUNCTION BUBBLESORT

```

fig04_17.cpp (8 of 8)

```
// OUTPUT ARRAY CONTENTS (20 VALUES PER ROW)
VOID PRINTARRAY( CONST INT A[], INT SIZE )
{
   FOR ( INT I = 0; I < SIZE; I++ ) {

      IF ( I % 20 == 0 ) // BEGIN NEW LINE EVERY 20 VALUES
         COUT << ENDL;

      COUT << SETW( 2 ) << A[ I ];

   } // END FOR

} // END FUNCTION PRINTARRAY
```

```
********

  MEAN

********

THE MEAN IS THE AVERAGE VALUE OF THE DATA

ITEMS. THE MEAN IS EQUAL TO THE TOTAL OF

ALL THE DATA ITEMS DIVIDED BY THE NUMBER

OF DATA ITEMS (99). THE MEAN VALUE FOR

THIS RUN IS: 681 / 99 = 6.8788

********

 MEDIAN

********

THE UNSORTED ARRAY OF RESPONSES IS

 6 7 8 9 8 7 8 9 8 9 7 8 9 5 9 8 7 8 7 8

 6 7 8 9 3 9 8 7 8 7 7 8 9 8 9 8 9 7 8 9

 6 7 8 7 8 7 9 8 9 2 7 8 9 8 9 8 9 7 5 3

 5 6 7 2 5 3 9 4 6 4 7 8 9 6 8 7 8 9 7 8

 7 4 4 2 5 3 8 7 5 6 4 5 6 1 6 5 7 8 7


THE SORTED ARRAY IS

 1 2 2 2 3 3 3 3 4 4 4 4 4 5 5 5 5 5 5 5

 5 6 6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7 7 7

 7 7 7 7 7 7 7 7 7 7 7 7 7 8 8 8 8 8 8 8

 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8

 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9
```

```
********

 MODE

********

RESPONSE  FREQUENCY        HISTOGRAM



                              1   1   2   2
                          5   0   5   0   5


     1        1        *
     2        3        ***
     3        4        ****
     4        5        *****
     5        8        ********
     6        9        *********
     7       23        ***********************
     8       27        ***************************
     9       19        *******************
```

THE MODE IS THE MOST FREQUENT VALUE.

FOR THIS RUN THE MODE IS 8 WHICH OCCURRED 27 TIMES.

4.8 Searching Arrays: Linear Search and Binary Search

Search array for a key value

Linear search

- Compare each element of array with key value
  - Start at one end, go to other
- Useful for small and unsorted arrays
  - Inefficient
  - If search key not present, examines every element

4.8 Searching Arrays: Linear Search and Binary Search

Binary search

- Only used with sorted arrays
- Compare middle element with key
  - If equal, match found
  - If key < middle
    - Repeat search on first half of array
  - If key > middle
    - Repeat search on last half
- Very fast
  - At most N steps, where 2 > # of elements
  - 30 element array takes at most 5 steps

    2 > 30

N

5

fig04_19.cpp (1 of 2)

Takes array, search key, and array size.

```
// FIG. 4.19: FIG04_19.CPP
// LINEAR SEARCH OF AN ARRAY.
#INCLUDE <IOSTREAM>

USING STD::COUT;
USING STD::CIN;
USING STD::ENDL;

INT LINEARSEARCH( CONST INT [], INT, INT );  // PROTOTYPE

```

fig04_19.cpp (2 of 2)

```
   // DISPLAY RESULTS
   IF ( ELEMENT != -1 )
      COUT << "FOUND VALUE IN ELEMENT " << ELEMENT << ENDL;
   ELSE
      COUT << "VALUE NOT FOUND" << ENDL;

   RETURN 0;  // INDICATES SUCCESSFUL TERMINATION

} // END MAIN

// COMPARE KEY TO EVERY ELEMENT OF ARRAY UNTIL LOCATION IS
// FOUND OR UNTIL END OF ARRAY IS REACHED; RETURN SUBSCRIPT OF
// ELEMENT IF KEY OR -1 IF KEY NOT FOUND
INT LINEARSEARCH( CONST INT ARRAY[], INT KEY, INT SIZEOFARRAY )
{
   FOR ( INT J = 0; J < SIZEOFARRAY; J++ )

      IF ( ARRAY[ J ] == KEY )  // IF FOUND,
         RETURN J;              // RETURN LOCATION OF KEY

   RETURN -1;  // KEY NOT FOUND

} // END FUNCTION LINEARSEARCH
```

fig04_19.cpp output (1 of 1)

```
ENTER INTEGER SEARCH KEY: 36
FOUND VALUE IN ELEMENT 18


ENTER INTEGER SEARCH KEY: 37
VALUE NOT FOUND
```

fig04_20.cpp (1 of 6)

```
// FIG. 4.20: FIG04_20.CPP
// BINARY SEARCH OF AN ARRAY.
#INCLUDE <IOSTREAM>

USING STD::COUT;
USING STD::CIN;
USING STD::ENDL;

#INCLUDE <IOMANIP>

USING STD::SETW;

// FUNCTION PROTOTYPES
INT BINARYSEARCH( CONST INT [], INT, INT, INT, INT );
VOID PRINTHEADER( INT );
VOID PRINTROW( CONST INT [], INT, INT, INT, INT );

INT MAIN()
{
   CONST INT ARRAYSIZE = 15;  // SIZE OF ARRAY A
   INT A[ ARRAYSIZE ];        // CREATE ARRAY A
   INT KEY;                   // VALUE TO LOCATE IN A

   FOR ( INT I = 0; I < ARRAYSIZE; I++ )  // CREATE SOME DATA
```

fig04_20.cpp (2 of 6)

```
   COUT << "ENTER A NUMBER BETWEEN 0 AND 28: ";
   CIN >> KEY;

   PRINTHEADER( ARRAYSIZE );

   // SEARCH FOR KEY IN ARRAY A
   INT RESULT =
      BINARYSEARCH( A, KEY, 0, ARRAYSIZE - 1, ARRAYSIZE );

   // DISPLAY RESULTS
   IF ( RESULT != -1 )
      COUT << '\N' << KEY << " FOUND IN ARRAY ELEMENT "
         << RESULT << ENDL;
   ELSE
      COUT << '\N' << KEY << " NOT FOUND" << ENDL;

   RETURN 0;  // INDICATES SUCCESSFUL TERMINATION

} // END MAIN

```

fig04_20.cpp (3 of 6)

```
// FUNCTION TO PERFORM BINARY SEARCH OF AN ARRAY
INT BINARYSEARCH( CONST INT B[], INT SEARCHKEY, INT LOW,
   INT HIGH, INT SIZE )
{
   INT MIDDLE;

   // LOOP UNTIL LOW SUBSCRIPT IS GREATER THAN HIGH SUBSCRIPT
   WHILE ( LOW <= HIGH ) {

      // DETERMINE MIDDLE ELEMENT OF SUBARRAY BEING SEARCHED
      MIDDLE = ( LOW + HIGH ) / 2;

      // DISPLAY SUBARRAY USED IN THIS LOOP ITERATION
      PRINTROW( B, LOW, MIDDLE, HIGH, SIZE );

```

Determine middle element

fig04_20.cpp (4 of 6)

```
      // IF SEARCHKEY MATCHES MIDDLE ELEMENT, RETURN MIDDLE
      IF ( SEARCHKEY == B[ MIDDLE ] )  // MATCH
         RETURN MIDDLE;

      ELSE

         // IF SEARCHKEY LESS THAN MIDDLE ELEMENT
         // SET NEW HIGH ELEMENT
         IF ( SEARCHKEY < B[ MIDDLE ] )
            HIGH = MIDDLE - 1;  // SEARCH LOW END O

         // IF SEARCHKEY GREATER THAN MIDDLE ELEM
         // SET NEW LOW ELEMENT
         ELSE
            LOW = MIDDLE + 1;   // SEARCH HIGH END OF ARRAY
   }

   RETURN -1;  // SEARCHKEY NOT FOUND

} // END FUNCTION BINARYSEARCH
```

Use the rule of binary search:
If key equals middle, match

If less, search low end

If greater, search high end

Loop sets low, middle and high dynamically. If searching the high end, the new low is the element above the middle.

fig04_20.cpp (5 of 6)

```

// PRINT HEADER FOR OUTPUT
VOID PRINTHEADER( INT SIZE )
{
   COUT << "\NSUBSCRIPTS:\N";

   // OUTPUT COLUMN HEADS
   FOR ( INT J = 0; J < SIZE; J++ )
      COUT << SETW( 3 ) << J << ' ';

   COUT << '\N'; // START NEW LINE OF OUTPUT

   // OUTPUT LINE OF - CHARACTERS
   FOR ( INT K = 1; K <= 4 * SIZE; K++ )
      COUT << '-';

   COUT << ENDL; // START NEW LINE OF OUTPUT

} // END FUNCTION PRINTHEADER

```

fig04_20.cpp (6 of 6)

```
// PRINT ONE ROW OF OUTPUT SHOWING THE CURRENT
// PART OF THE ARRAY BEING PROCESSED
VOID PRINTROW( CONST INT B[], INT LOW, INT MID,
   INT HIGH, INT SIZE )
{
   // LOOP THROUGH ENTIRE ARRAY
   FOR ( INT M = 0; M < SIZE; M++ )

      // DISPLAY SPACES IF OUTSIDE CURRENT SUBARRAY RANGE
      IF ( M < LOW || M > HIGH )
```

fig04_20.cpp output (1 of 2)

```
ENTER A NUMBER BETWEEN 0 AND 28: 6

SUBSCRIPTS:
   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
------------------------------------------------------------
   0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
   0   2   4   6*  8  10  12

6 FOUND IN ARRAY ELEMENT 3
```

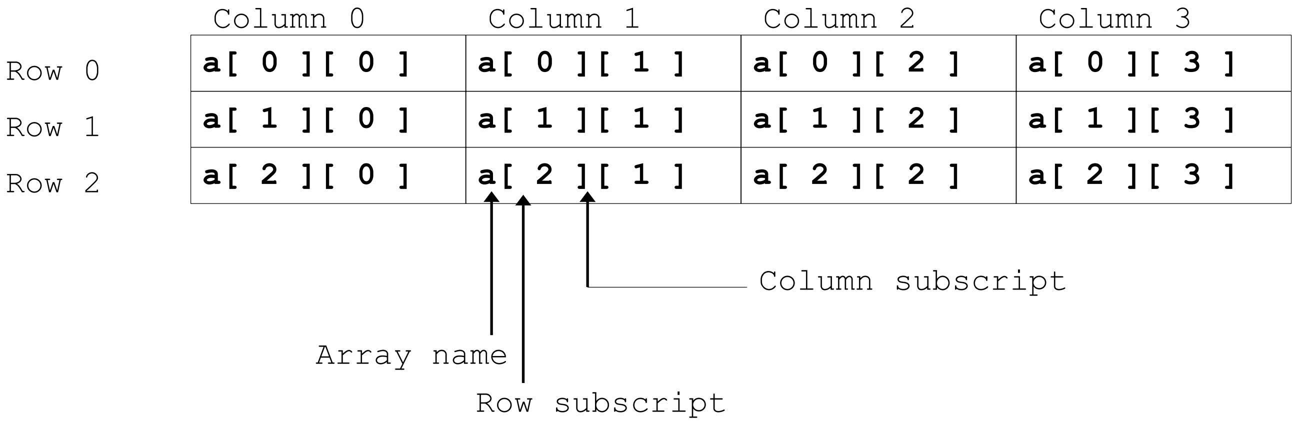
```
ENTER A NUMBER BETWEEN 0 AND 28: 8

SUBSCRIPTS:
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14
------------------------------------------------------------
  0  2  4  6  8 10 12 14* 16 18 20 22 24 26 28
  0  2  4  6*  8 10 12
             8 10* 12
             8*

8 FOUND IN ARRAY ELEMENT 4
```

fig04_20.cpp output (2 of 2)

4.9 Multiple-Subscripted Arrays

Multiple subscripts

- **`a[ i ][ j ]`**
- Tables with rows and columns
- Specify row, then column
- “Array of arrays”
  - **`a[0]`** is an array of 4 elements
  - **`a[0][0]`** is the first element of that array



	Column 0	Column 1	Column 2	Column 3
Row 0	**a[0][0]**	**a[0][1]**	**a[0][2]**	**a[0][3]**
Row 1	**a[1][0]**	**a[1][1]**	**a[1][2]**	**a[1][3]**
Row 2	**a[2][0]**	**a[2][1]**	**a[2][2]**	**a[2][3]**

Column subscript

Array name

Row subscript

4.9 Multiple-Subscripted Arrays

To initialize

- Default of 0
- Initializers grouped by row in braces

```
int b[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };
```

Row 0 Row 1

1	2
3	4

```
int b[ 2 ][ 2 ] = { { 1 }, { 3, 4 } };
```

1	0
3	4

4.9 Multiple-Subscripted Arrays

1	0
3	4

Referenced like normal

```
cout << b[ 0 ][ 1 ];
```

- Outputs 0
- Cannot reference using commas

  ```
  cout << b[ 0, 1 ];
  ```

  - Syntax error

Function prototypes

- Must specify sizes of subscripts
  - First subscript not necessary, as with single-scripted arrays
- `void printArray( int [][ 3 ] );`

fig04_22.cpp (1 of 2)

```
// FIG. 4.22: FIG04_22.CPP
// INITIALIZING MULTIDIMENSIONAL ARRAYS.
#INCLUDE <IOSTREAM>

USING STD::COUT;
USING STD::ENDL;

VOID PRINTARRAY( INT [][ 3 ] );

INT MAIN()
{
  INT ARRAY1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } };
  INT ARRAY2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 };
  INT ARRAY3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };

  COUT << "VALUES IN ARRAY1 BY ROW ARE:" << ENDL;
  PRINTARRAY( ARRAY1 );

  COUT << "VALUES IN ARRAY2 BY ROW ARE:" << ENDL;
  PRINTARRAY( ARRAY2 );

  COUT << "VALUES IN ARRAY3 BY ROW ARE:" << ENDL;
  PRINTARRAY( ARRAY3 );
```

Note the format of the prototype.

Note the various initialization styles. The elements in **`array2`** are assigned to the first row and then the second.

fig04_22.cpp output (1 of 1)

```

// FUNCTION TO OUTPUT ARRAY WITH TWO ROW
VOID PRINTARRAY( INT A[][ 3 ] )
{
   FOR ( INT I = 0; I < 2; I++ ) {   // FOR EACH ROW

      FOR ( INT J = 0; J < 3; J++ )  // OUTPUT COLUMN VALUES
         COUT << A[ I ][ J ] << ' ';

      COUT << ENDL;  // START NEW LINE OF OUTPUT

```

For loops are often used to iterate through arrays. Nested loops are helpful with multiple-subscripted arrays.

```
Values in array1 by row are:
1 2 3
4 5 6
Values in array2 by row are:
1 2 3
4 5 0
Values in array3 by row are:
1 2 0
4 0 0
```

4.9 Multiple-Subscripted Arrays

Next: program showing initialization

- After, program to keep track of students grades
- Multiple-subscripted array (table)
- Rows are students
- Columns are grades

	Quiz1	Quiz2
Student0	95	85
Student1	89	80

fig04_23.cpp (1 of 6)

```
// FIG. 4.23: FIG04_23.CPP
// DOUBLE-SUBSCRIPTED ARRAY EXAMPLE.
#INCLUDE <IOSTREAM>

USING STD::COUT;
USING STD::ENDL;
USING STD::FIXED;
USING STD::LEFT;

#INCLUDE <IOMANIP>

USING STD::SETW;
USING STD::SETPRECISION;

CONST INT STUDENTS = 3;   // NUMBER OF STUDENTS
CONST INT EXAMS = 4;      // NUMBER OF EXAMS

// FUNCTION PROTOTYPES
INT MINIMUM( INT [][ EXAMS ], INT, INT );
INT MAXIMUM( INT [][ EXAMS ], INT, INT );
DOUBLE AVERAGE( INT [], INT );
VOID PRINTARRAY( INT [][ EXAMS ], INT, INT );

```

fig04_23.cpp (2 of 6)

```
INT MAIN()
{
   // INITIALIZE STUDENT GRADES FOR THREE STUDENTS (ROWS)
   INT STUDENTGRADES[ STUDENTS ][ EXAMS ] =
      { { 77, 68, 86, 73 },
        { 96, 87, 89, 78 },
        { 70, 90, 86, 81 } };

   // OUTPUT ARRAY STUDENTGRADES
   COUT << "THE ARRAY IS:\N";
   PRINTARRAY( STUDENTGRADES, STUDENTS, EXAMS );

   // DETERMINE SMALLEST AND LARGEST GRADE VALUES
   COUT << "\N\NLOWEST GRADE: "
        << MINIMUM( STUDENTGRADES, STUDENTS, EXAMS )
        << "\NHIGHEST GRADE: "
        << MAXIMUM( STUDENTGRADES, STUDENTS, EXAMS ) << '\N';

   COUT << FIXED << SETPRECISION( 2 );

```

fig04_23.cpp (3 of 6)

```
  // CALCULATE AVERAGE GRADE FOR EACH STUDENT
  FOR ( INT PERSON = 0; PERSON < STUDENTS; PERSON++ )
    COUT << "THE AVERAGE GRADE FOR STUDENT " << PERSON
         << " IS "
         << AVERAGE( STUDENTGRADES[ PERSON ], EXAMS )
         << ENDL;

  RETURN 0;  // INDICATES SUCCESSFUL TERMINATIO

} // END MAIN

// FIND MINIMUM GRADE
INT MINIMUM( INT GRADES[][ EXAMS ], INT PUPILS, INT TESTS )
{
  INT LOWGRADE = 100; // INITIALIZE TO HIGHEST POSSIBLE GRADE


  FOR ( INT I = 0; I < PUPILS; I++ )

    FOR ( INT J = 0; J < TESTS; J++ )

      IF ( GRADES[ I ][ J ] < LOWGRADE )
        LOWGRADE = GRADES[ I ][ J ];


  RETURN LOWGRADE;
```

Determines the average for one student. We pass the array/row containing the student’s grades. Note that **`studentGrades[0]`** is itself an array.

fig04_23.cpp (4 of 6)

```
// FIND MAXIMUM GRADE
INT MAXIMUM( INT GRADES[][ EXAMS ], INT PUPILS, INT TESTS )
{
   INT HIGHGRADE = 0;  // INITIALIZE TO LOWEST POSSIBLE GRADE

   FOR ( INT I = 0; I < PUPILS; I++ )

      FOR ( INT J = 0; J < TESTS; J++ )

         IF ( GRADES[ I ][ J ] > HIGHGRADE )
            HIGHGRADE = GRADES[ I ][ J ];

   RETURN HIGHGRADE;

} // END FUNCTION MAXIMUM

```

fig04_23.cpp (5 of 6)

```
// DETERMINE AVERAGE GRADE FOR PARTICULAR STUDENT
DOUBLE AVERAGE( INT SETOFGRADES[], INT TESTS )
{
   INT TOTAL = 0;

   // TOTAL ALL GRADES FOR ONE STUDENT
   FOR ( INT I = 0; I < TESTS; I++ )
      TOTAL += SETOFGRADES[ I ];

   RETURN STATIC_CAST< DOUBLE >( TOTAL ) / TESTS;  // AVERAGE

} // END FUNCTION MAXIMUM
```

fig04_23.cpp (6 of 6)

```

// PRINT THE ARRAY
VOID PRINTARRAY( INT GRADES[][ EXAMS ], INT PUPILS, INT TESTS )
{
  // SET LEFT JUSTIFICATION AND OUTPUT COLUMN HEADS
  COUT << LEFT << "                 [0]  [1]  [2]  [3]";

  // OUTPUT GRADES IN TABULAR FORMAT
  FOR ( INT I = 0; I < PUPILS; I++ ) {

    // OUTPUT LABEL FOR ROW
    COUT << "\NSTUDENTGRADES[" << I << "] ";

    // OUTPUT ONE GRADES FOR ONE STUDENT
    FOR ( INT J = 0; J < TESTS; J++ )
      COUT << SETW( 5 ) << GRADES[ I ][ J ];

  } // END OUTER FOR

} // END FUNCTION PRINTARRAY
```

```
THE ARRAY IS:
                    [0]  [1]  [2]  [3]
STUDENTGRADES[0]    77   68   86   73
STUDENTGRADES[1]    96   87   89   78
STUDENTGRADES[2]    70   90   86   81


LOWEST GRADE: 68
HIGHEST GRADE: 96
THE AVERAGE GRADE FOR STUDENT 0 IS 76.00
THE AVERAGE GRADE FOR STUDENT 1 IS 87.50
THE AVERAGE GRADE FOR STUDENT 2 IS 81.75
```