# Introduction

- Divide and conquer
  - Construct a program from smaller pieces or components
  - Each piece more manageable than the original program
5.2  Program Modules in C

- Functions
  - Modules in C
  - Programs written by combining user-defined functions with library functions
    - C standard library has a wide variety of functions
    - Makes programmer's job easier - avoid reinventing the wheel

- Function calls
  - Invoking functions
    - Provide function name and arguments (data)
    - Function performs operations or manipulations
    - Function returns results
  - Boss asks worker to complete task
    - Worker gets information, does task, returns result
    - Information hiding: boss does not know details

5.3  Math Library Functions

- Math library functions
  - perform common mathematical calculations
  - #include <math.h>

- Format for calling functions

  FunctionName (argument);
  - If multiple arguments, use comma-separated list

  printf("%.2f", sqrt(900.0));
  - Calls function sqrt, which returns the square root of its argument
  - All math functions return data type double
  - Arguments may be constants, variables, or expressions
5.4 Functions

- Functions
  - Modularize a program
  - All variables declared inside functions are local variables
    - Known only in function defined
  - Parameters
    - Communicate information between functions
    - Local variables

- Benefits
  - Divide and conquer
    - Manageable program development
  - Software reusability
    - Use existing functions as building blocks for new programs
    - Abstraction - hide internal details (library functions)
  - Avoids code repetition

5.5 Function Definitions

- Function definition format

  \[
  \text{return-value-type \ function-name(} \ \text{parameter-list \ )} \\
  \{ \\
  \ \text{declarations and statements} \\
  \}
  \]

  - Function-name: any valid identifier
  - Return-value-type: data type of the result (default \textit{int})
    - \textbf{void} - function returns nothing
  - Parameter-list: comma separated list, declares parameters (default \textit{int})
5.5 Function Definitions (II)

- Function definition format (continued)

```
return-value-type function-name ( parameter-list )
{
    declarations and statements
}
```

- Declarations and statements: function body (block)
  - Variables can be declared inside blocks (can be nested)
  - Function can not be defined inside another function

- Returning control
  - If nothing returned
    - `return;`
  - or, until reaches right brace
  - If something returned
    - `return expression;`

```
#include <stdio.h>

int maximum( int, int, int ); /* function prototype */

int main()
{
    int a, b, c;
    printf( "Enter three integers: " );
    scanf( "%d%d%d", &a, &b, &c );
    printf( "Maximum is: %d\n", maximum( a, b, c ) );
    return 0;
}

/* Function maximum definition */
int maximum( int x, int y, int z )
{
    int max = x;
    if ( y > max )
        max = y;
    if ( z > max )
        max = z;
    return max;
}
```

Enter three integers: 22 85 17
Maximum is: 85
5.6 Function Prototypes

- Function prototype
  - Function name
  - Parameters - what the function takes in
  - Return type - data type function returns (default `int`)
  - Used to validate functions
  - Prototype only needed if function definition comes after use in program
    ```
    int maximum(int, int, int);
    ```
    - Takes in 3 `ints`
    - Returns an `int`

- Promotion rules and conversions
  - Converting to lower types can lead to errors

5.7 Header Files

- Header files
  - contain function prototypes for library functions
  - `<stdlib.h>, <math.h>`, etc
  - Load with `#include <filename>`
    ```
    #include <math.h>
    ```

- Custom header files
  - Create file with functions
  - Save as `filename.h`
  - Load in other files with `#include "filename.h"`
  - Reuse functions
5.8 Calling Functions: Call by Value and Call by Reference

- Used when invoking functions
- Call by value
  - Copy of argument passed to function
  - Changes in function do not effect original
  - Use when function does not need to modify argument
    - Avoids accidental changes
- Call by reference
  - Passes original argument
  - Changes in function effect original
  - Only used with trusted functions
- For now, we focus on call by value

5.9 Random Number Generation

- \texttt{rand} function
  - Load \texttt{<stdlib.h>}
  - Returns "random" number between 0 and \texttt{RAND_MAX} (at least 32767)
    \begin{verbatim}
    i = rand();
    \end{verbatim}
  - Pseudorandom
    - Preset sequence of "random" numbers
    - Same sequence every time program is executed
- Scaling
  - To get a random number between 1 and \texttt{n}
    \begin{verbatim}
    1 + ( \texttt{rand()} \% \texttt{n} )
    \end{verbatim}
  - \texttt{rand} \% \texttt{n} returns a number between 0 and \texttt{n-1}
  - Add 1 to make random number between 1 and \texttt{n}
    \begin{verbatim}
    1 + ( \texttt{rand()} \% 6 ) /* number between 1 and 6 */
    \end{verbatim}
5.9 Random Number Generation (II)

- **srand function**
  - `<stdlib.h>`
  - Takes an integer seed - jumps to location in "random" sequence
    
    ```c
    srand( seed );
    ```
  -  ```c
    srand( time( NULL ) ); /* load <time.h> */
    ```
  -  ```c
    time( NULL ) - time program was compiled in seconds
    ```
  - "randomizes" the seed

```c
1 /* Fig. 5.9: fig05_09.c */
2 #include <stdlib.h>
3 #include <stdio.h>
4
5 int main()
6 {
7    int i;
8    unsigned seed;
9
10    printf( "Enter seed: " );
11    scanf( "%u", &seed );
12    srand( seed );
13
14    for ( i = 1; i <= 10; i++ ) {
15        printf( "%10d", 1 + ( rand() % 6 ) );
16        if ( i % 5 == 0 )
17            printf( "\n" );
18    }
19
20    return 0;
21 }
```
5.10 Example: A Game of Chance

- Craps simulator
- Rules
  - Roll two dice
    - 7 or 11 on first throw, player wins
    - 2, 3, or 12 on first throw, player loses
    - 4, 5, 6, 8, 9, 10 - value becomes player's "point"
  - Player must roll his point before rolling 7 to win
  - Player loses if he rolls 7 before rolling his point
# Craps

## 1. rollDice

**Prototype**

```c
int rollDice( void );
```

## 1.1 Initialize variables

```c
int gameStatus, sum, myPoint;
```

## 1.2 Seed srand

```c
srand( time( NULL ) );
```

## 2. Define switch statement for win/loss/continue

```c
switch ( sum ) {
    case 7: case 11: /* win on first roll */
        gameStatus = 1;
        break;

    case 2: case 3: case 12: /* lose on first roll */
        gameStatus = 2;
        break;

    default: /* remember point */
        gameStatus = 0;
        myPoint = sum;
        printf( "Point is %d\n", myPoint );
        break;
}
```

## 2.1 Loop

```c
while ( gameStatus == 0 ) { /* keep rolling */
    sum = rollDice();
}
```

## 2.2 Print win/loss

```c
if ( gameStatus == 1 )
    printf( "Player wins\n" );
else
    printf( "Player loses\n" );
return 0;
```

## 3. rollDice

```c
int rollDice( void )
```

```c
{ int die1, die2, workSum;

    die1 = 1 + ( rand() % 6 );
    die2 = 1 + ( rand() % 6 );
    workSum = die1 + die2;
    printf( "Player rolled %d + %d = %d\n", die1, die2, workSum );
    return workSum;
}
```

**Program Output**

```
Player rolled 6 + 5 = 11
Player wins
```
**5.11 Storage Classes**

- **Storage class specifiers**
  - Storage duration - how long an object exists in memory
  - Scope - where object can be referenced in program
  - Linkage - what files an identifier is known (more in Chapter 14)

- **Automatic storage**
  - Object created and destroyed within its block
  - `auto`: default for local variables
    - `auto double x, y;`
  - `register`: tries to put variable into high-speed registers
    - Can only be used for automatic variables
    - `register int counter = 1;`
5.11 Storage Classes (II)

- Static storage
  - Variables exist for entire program execution
  - Default value of zero
  - `static`: local variables defined in functions.
    - Keep value after function ends
    - Only known in their own function.
  - `extern`: default for global variables and functions.
    - Known in any function

5.12 Scope Rules

- File scope
  - Identifier defined outside function, known in all functions
  - Global variables, function definitions, function prototypes

- Function scope
  - Can only be referenced inside a function body
  - Only labels (`start: case:`, etc.)
5.12 Scope Rules (II)

- Block scope
  - Identifier declared inside a block
    - Block scope begins at declaration, ends at right brace
  - Variables, function parameters (local variables of function)
  - Outer blocks "hidden" from inner blocks if same variable name

- Function prototype scope
  - Identifiers in parameter list
  - Names in function prototype optional, and can be used anywhere

```
/* Fig. 5.12: fig05_12.c */
#include <stdio.h>

void a(void); /* function prototype */
void b(void); /* function prototype */
void c(void); /* function prototype */

int x = 1; /* global variable */

int main()

int x = 5; /* local variable to main */

printf("local x in outer scope of main is %d\n", x);

int x = 7;

printf("local x in inner scope of main is %d\n", x);

} /* end new scope */

printf("local x in outer scope of main is %d\n", x);

a(); /* a has automatic local x */
b(); /* b has static local x */
c(); /* c uses global x */
a(); /* a reinitializes automatic local x */
b(); /* static local x retains its previous value */
c(); /* global x also retains its value */
```
31    printf( "local x in main is %d\n", x );
32    return 0;
33 }
34
35 void a( void )
36 {
37    int x = 25; /* initialized each time a is called */
38    printf( "local x in a is %d after entering a\n", x );
39    ++x;
40    printf( "local x in a is %d before exiting a\n", x );
41 }
42
43 void b( void )
44 {
45    static int x = 50; /* static initialization only */
46    /* first time b is called */
47    printf( "local static x is %d on entering b\n", x );
48    ++x;
49    printf( "local static x is %d on exiting b\n", x );
50 }
51
52 void c( void )
53 {
54    printf( "global x is %d on entering c\n", x );
55    x *= 10;
56    printf( "global x is %d on exiting c\n", x );
57 }
58
© 2000 Prentice Hall, Inc. All rights reserved.
5.13 Recursion

- Recursive functions
  - Function that calls itself
  - Can only solve a base case
  - Divides up problem into
    - What it can do
    - What it cannot do - resembles original problem
      - Launches a new copy of itself (recursion step)

- Eventually base case gets solved
  - Gets plugged in, works its way up and solves whole problem

5.13 Recursion (II)

- Example: factorial:
  \[ 5! = 5 \times 4 \times 3 \times 2 \times 1 \]
  Notice that
  \[ 5! = 5 \times 4! \]
  \[ 4! = 4 \times 3! \ldots \]
  - Can compute factorials recursively
  - Solve base case (1! = 0! = 1) then plug in
    - 2! = 2 \times 1! = 2 \times 1 = 2
    - 3! = 3 \times 2! = 3 \times 2 = 6
5.14 Example Using Recursion: The Fibonacci Series

- Fibonacci series: 0, 1, 1, 2, 3, 5, 8...
  - Each number sum of the previous two

\[ \text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2) \] - recursive formula

```c
long fibonacci(long n)
{
    if (n==0 || n==1) //base case
        return n;
    else return fibonacci(n-1) + fibonacci(n-2);
}
```

5.14 Example Using Recursion: The Fibonacci Series (II)

```
f(3)

return f(2) + f(1)

return f(1) + f(0)

return 1

return 0

return 1
```

© 2000 Prentice Hall, Inc. All rights reserved.
1 /* Fig. 5.15: fib05_15.c */
2 #include <stdio.h>
3
4 long fibonacci( long );
5
6 int main()
7 {
8     long result, number;
9
10    printf( "Enter an integer: " );
11    scanf( "%ld", &number );
12    result = fibonacci( number );
13    printf( "Fibonacci(%ld) = %ld\n", number, result );
14    return 0;
15 }
16
17 /* Recursive definition of function fibonacci */
18 long fibonacci( long n )
19 {
20    if ( n == 0 || n == 1 )
21        return n;
22    else
23        return fibonacci(n - 1) + fibonacci(n - 2);
24 }

Enter an integer: 0
Fibonacci(0) = 0

Enter an integer: 1
Fibonacci(1) = 1

Enter an integer: 2
Fibonacci(2) = 1

Enter an integer: 3
Fibonacci(3) = 2

Enter an integer: 4
Fibonacci(4) = 3

Enter an integer: 5
Fibonacci(5) = 5

Enter an integer: 6
Fibonacci(6) = 8

Enter an integer: 10
Fibonacci(10) = 55

Enter an integer: 20
Fibonacci(20) = 6765

Enter an integer: 30
Fibonacci(30) = 832040

Enter an integer: 35
Fibonacci(35) = 9227465
5.15 Recursion vs. Iteration

- Repetition
  - Iteration: explicit loop
  - Recursion: repeated function calls

- Termination
  - Iteration: loop condition fails
  - Recursion: base case recognized

- Both can have infinite loops

- Balance
  - Choice between performance (iteration) and good software engineering (recursion)

Chapter 6 - Arrays

Outline

6.1 Introduction
6.2 Arrays
6.3 Declaring Arrays
6.4 Examples Using Arrays
6.5 Passing Arrays to Functions
6.6 Sorting Arrays
6.7 Computing Mean, Median and Mode Using Arrays
6.8 Searching Arrays
6.9 Multiple-Subscripted Arrays
6.1 Introduction

- **Arrays**
  - Structures of related data items
  - Static entity - same size throughout program
  - Dynamic data structures discussed in Chapter 12

6.2 Arrays

- **Array**
  - Group of consecutive memory locations
  - Same name and type

- To refer to an element, specify
  - Array name
  - Position number

- Format: `arrayname[position number]`
  - First element at position 0
  - n element array named `c: c[0], c[1]...c[n-1]`
6.2 Arrays (II)

- Array elements are like normal variables
  \[ c[0] = 3; \]
  \[ \text{printf}( \"%d\", c[0] ); \]

  - Perform operations in subscript. If \( x = 3 \),

6.3 Declaring Arrays

- When declaring arrays, specify
  - Name
  - Type of array
  - Number of elements
    \[ \text{arrayType arrayName[ numberOfElements ]}; \]
    \[ \text{int c[ 10 ]}; \]
    \[ \text{float myArray[ 3284 ]}; \]

- Declaring multiple arrays of same type
  - Format similar to regular variables
    \[ \text{int b[ 100 ], x[ 27 ]}; \]
6.4 Examples Using Arrays

- Initializers
  int n[5] = {1, 2, 3, 4, 5};
  - If not enough initializers, rightmost elements become 0
  - If too many, syntax error
    int n[5] = {0}
    • All elements 0
  - C arrays have no bounds checking

- If size omitted, initializers determine it
  int n[] = {1, 2, 3, 4, 5};
  - 5 initializers, therefore 5 element array

```c
1 /* Fig. 6.8: fig06_08.c */
2 #include <stdio.h>
3 #define SIZE 10
4
5 int main()
6 {
7    int n[SIZE] = {19, 3, 15, 7, 11, 9, 13, 5, 17, 1};
8    int i, j;
9
10    printf( "%s\%3s\%17s\n", "Element", "Value", "Histogram" );
11    for ( i = 0; i <= SIZE - 1; i++ ) {
12        printf( "%d\%13d         ", i, n[i] );
13        for ( j = 1; j <= n[i]; j++ ) /* print one bar */
14            printf( "\%c", '\*';
15        printf( "\n" );
16    }
17    return 0;
18 }
```

© 2000 Prentice Hall, Inc. All rights reserved.
### 6.4 Examples Using Arrays (II)

- **Character arrays**
  - String "hello" is really a `static` array of characters
  - Character arrays can be initialized using string literals
    ```
    char string1[] = "first";
    ```
    - null character ‘\0’ terminates strings
    - `string1` actually has 6 elements
    ```
    char string1[] = { 'f', 'i', 'r', 's', 't', '\0' };
    ```
6.4 Examples Using Arrays (III)

- Character arrays (continued)
  - Access individual characters
    * `string1[3]` is character 's'
  - Array name is address of array, so & not needed for `scanf`
    ```
    scanf("%s", string2);
    ```
    * Reads characters until whitespace encountered
    * Can write beyond end of array, be careful

```c
/* Fig. 6.10: fig06_10.c
   Treating character arrays as strings */
#include <stdio.h>

int main()
{
    char string1[20], string2[] = "string literal";
    int i;

    printf("Enter a string: ");
    scanf("%s", string1);
    printf("string1 is: %s
string2 is: %s
string1 with spaces between characters is:
", string1, string2);

    for ( i = 0; string1[ i ] != '\0'; i++ )
        printf("%c", string1[ i ]); 

    printf("\n");
    return 0;
}
```

Enter a string: Hello there
string1 is: Hello
string2 is: string literal
string1 with spaces between characters is:
Hello there
6.5 Passing Arrays to Functions

• Passing arrays
  – Specify array name without brackets
    
    ```c
    int myArray[ 24 ];
    myFunction( myArray, 24 );
    ```
  – Array size usually passed to function
  – Arrays passed call-by-reference
  – Name of array is address of first element
  – Function knows where the array is stored
    • Modifies original memory locations

• Passing array elements
  – Passed by call-by-value
  – Pass subscripted name (i.e., `myArray[3]`) to function

6.5 Passing Arrays to Functions (II)

• Function prototype
  
  ```c
  void modifyArray( int b[], int arraySize );
  ```
  – Parameter names optional in prototype
    • `int b[]` could be simply `int []`
    • `int arraySize` could be simply `int`
1 /* Fig. 6.13: fig06_13.c */
2 Passing arrays and individual array elements to functions */
3 #include <stdio.h>
4 #define SIZE 5
5
6 void modifyArray( int [], int ); /* appears strange */
7 void modifyElement( int );
8
9 int main()
10 {
11    int a[ SIZE ] = { 0, 1, 2, 3, 4 }, i;
12
13    printf( "Effects of passing entire array call "
14        "by reference:\n\nThe values of the "
15        "original array are:\n" );
16
17    for ( i = 0; i <= SIZE - 1; i++ )
18       printf( " %3d " , a[ i ] );
19
20    printf( "\n" );
21    modifyArray( a, SIZE ); /* passed call by reference */
22    printf( "The values of the modified array are:\n" );
23
24    for ( i = 0; i <= SIZE - 1; i++ )
25       printf( " %3d " , a[ i ] );
26
27    printf( "\n\nEffects of passing array element call "
28        "by value:\n\n\nThe values of a[3] is %d\n", a[ 3 ] );
29    modifyElement( a[ 3 ] );
30    printf( "The value of a[3] is %d\n", a[ 3 ] );
31    return 0;
32 }

33 void modifyArray( int b[], int size )
34 {
35    int j;
36
37    for ( j = 0; j <= size - 1; j++ )
38       b[ j ] *= 2;
39 }
40
41 void modifyElement( int e )
42 {
43    printf( "Value in modifyElement is %d\n", e *= 2 );
44 }
45 }

Effects of passing entire array call 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 call by value:

The value of a[3] is 6
Value in modifyElement is 12
The value of a[3] is 6
6.6 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
- Example:
  - Original: 3 4 2 6 7
  - Pass 1: 3 2 4 6 7
  - Pass 2: 2 3 4 6 7
  - Small elements "bubble" to the top

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

- Mean - average
- Median - number in middle of sorted list
  - 1, 2, 3, 4, 5
  - 3 is the median
- Mode - number that occurs most often
  - 1, 1, 1, 2, 3, 3, 4, 5
  - 1 is the mode
This program introduces the topic of survey data analysis.
It computes the mean, median, and mode of the data */
#include <stdio.h>
#define SIZE 99

void mean( const int [] );
void median( int [] );
void mode( int [], const int [] );
void bubbleSort( int [] );
void printArray( const int [] );

int main()
{
    int frequency[ 10 ] = { 0 };
    int response[ SIZE ] =
    { 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, 9, 7, 8, 9, 7,
    7, 8, 9, 8, 9, 8, 9, 8, 9, 2,
    5, 6, 7, 2, 5, 3, 9, 4, 6, 4,
    7, 8, 9, 6, 8, 7, 8, 9, 7, 8,
    7, 4, 2, 5, 3, 8, 7, 5, 6,
    4, 5, 6, 1, 6, 5, 7, 8, 7 };
    mean( response );
    median( response );
    mode( frequency, response );
    return 0;
}

void mean( const int answer[] )
{
    int j, total = 0;
    printf( "\n\n\n*****\n* Mean *\n*****\n\n" );
    for ( j = 0; j <= SIZE - 1; j++ )
        total += answer[ j ];
    printf( "The mean is the average value of the data items. The mean is equal to the total of\n" );
    printf( "the data items divided by the number\n" );
    printf( "of data items (\%d\). The mean value for\n" );
    printf( "this run is: \%d / \%d = \%.4f\n\n", SIZE, total, SIZE, ( double ) total / SIZE );
}

void median( int answer[] )
{
    printf( "\n\n\n*****\n* Median *\n*****\n\n" );
    "The unsorted array of responses is" ;
    printf( "\n\n\n*****\n* Median *\n*****\n\n" );
    printf( "\n\n\n*****\n* Median *\n*****\n\n" );
    printf( "The unsorted element array is" );
    printArray( answer );
    bubbleSort( answer );
    printf( "\n\n\n*****\n* Median *\n*****\n\n" );
    printf( "\n\n\n*****\n* Median *\n*****\n\n" );
    printf( "The sorted element array\n" );
    printf( "For this run the median is \%d of\n" );
    printf( "the sorted \%d element array\n" );
    printArray( answer );
    printf( "\n\n\n*****\n* Median *\n*****\n\n" );
    printf( "\n\n\n*****\n* Median *\n*****\n\n" );
    printf( "For this run the median is \%d of\n" );
    printf( "the sorted \%d element array\n" );
    printf( "For this run the median is \%d of\n" );
    printf( "the sorted \%d element array\n" );
3.2 Define function `mode`

3.2.1 Increase frequency[] depending on `response[]`

Notice how the subscript in `frequency[]` is the value of an element in `response[]` (answer[]). Print stars depending on value of `frequency[]`

3.3 Define `bubbleSort`

3.3 Define `printArray`

Bubble sort: if elements are out of order, swap them.
126    printf("%2d", a[ j ] );
127 }
128 }
129

********
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
7 8 9 8 9 8 9 9 8 7 8 8 9 8 9 7 8 9
6 7 8 9 7 8 9 8 2 7 8 9 8 9 8 7 5 3
5 6 7 5 3 9 4 6 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 4 4 4 4 4 5 5 5 5 5 5
5 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 0 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

The median is element 49 of the sorted 99 element array.
For this run the median is 7

********
Mode
********
Response  Frequency  Histogram
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.
6.8 Searching Arrays: Linear Search and Binary Search

• Search an array for a key value

• Linear search
  – Simple
  – Compare each element of array with key value
  – Useful for small and unsorted arrays

• Binary search
  – For sorted arrays
  – Compares middle element with key
    • If equal, match found
    • If key < middle, looks in first half of array
    • If key > middle, looks in last half
    • Repeat
  – Very fast; at most \( n \) steps, where \( 2^n > \) number of elements
    • 30 element array takes at most 5 steps
      \( 2^5 > 30 \)
6.9 Multiple-Subscripted Arrays

- Multiple subscripted arrays
  - Tables with rows and columns ($m$ by $n$ array)
  - Like matrices: specify row, then column

<table>
<thead>
<tr>
<th>Row 0</th>
<th>Column 0</th>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a[0][0]</td>
<td>a[0][1]</td>
<td>a[0][2]</td>
<td>a[0][3]</td>
</tr>
<tr>
<td>Row 1</td>
<td>a[1][0]</td>
<td>a[1][1]</td>
<td>a[1][2]</td>
<td>a[1][3]</td>
</tr>
</tbody>
</table>

Array name

Row subscript

Column subscript

© 2000 Prentice Hall, Inc. All rights reserved

6.9 Multiple-Subscripted Arrays (II)

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

  Initializers grouped by row in braces
  
  If not enough, unspecified elements set to zero

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

- Referencing elements
  
  ```
  printf( "%d", b[0][1] );  
  ```

  Specify row, then column
```c
/* Fig. 6.22: fig06_22.c */
double-subscripted array example */
#include <stdio.h>
#define STUDENTS 3
#define EXAMS 4
int minimum[const int][EXAMS], int, int);
int maximum[const int][EXAMS], int, int);
double average(const int [], int );
void printArray(const int [], int, int );
int main(){
    int student;
    const int studentGrades[STUDENTS][EXAMS] =
        { { 77, 68, 86, 73 },
          { 96, 87, 89, 78 },
          { 70, 90, 86, 81 } };
    printf("The array is:\n");
    printArray(studentGrades, STUDENTS, EXAMS);
    printf("\n\nLowest grade: %d\nHighest grade: %d\n",
        minimum(studentGrades, STUDENTS, EXAMS),
        maximum(studentGrades, STUDENTS, EXAMS));
    for (student = 0; student < STUDENTS - 1; student++)
        printf("The average grade for student %d is %.2f\n",
            student,
            average(studentGrades[student], EXAMS));
    return 0;
}

/* Find the minimum grade */
int minimum(const int grades[][EXAMS],
            int pupils, int tests )
{
    int i, j, lowGrade = 100;
    for (i = 0; i < pupils - 1; i++)
        for (j = 0; j < tests - 1; j++)
            if (grades[i][j] < lowGrade)
                lowGrade = grades[i][j];
    return lowGrade;
}

/* Find the maximum grade */
int maximum(const int grades[][EXAMS],
            int pupils, int tests )
{
    int i, j, highGrade = 0;
    for (i = 0; i < pupils - 1; i++)
        for (j = 0; j < tests - 1; j++)
            if (grades[i][j] > highGrade)
                highGrade = grades[i][j];
    return highGrade;
}

/* Determine the average grade for a particular exam */
double average(const int setOfGrades[], int tests )
{
int i, total = 0;
for (i = 0; i < tests - 1; i++)
    total += setOfGrades[ i ];
return (double) total / tests;

/* Print the array */
void printArray( const int grades[][ EXAMS ],
        int pupils, int tests )
{
    int i, j;

    printf( "
[0] [1] [2] [3] ");
    for ( i = 0; i < pupils - 1; i++ )
        printf( "studentGrades[%d ], i);
    for ( j = 0; j < tests - 1; j++ )
        printf( "%5d", grades[ i ][ j ]);
}

The array is:

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