Lecture 04

FUNCTIONS AND ARRAYS

Motivations

- Divide hug tasks to blocks: divide programs up into sets of cooperating functions.
- Define new functions with function calls and parameter passing.
- Use functions to reduce code duplication and increase program modularity.
- Easy to:
  - Design → Implement → Test → Maintain → Reuse

Function of Functions

- Decomposition and abstraction through functions
  - Break up into modules
  - Suppress detail
  - Create “new primitive”

Functions, Informally

- A function is like a *subprogram*, a small program inside of a program.
- The basic idea – we write a sequence of statements and then give that sequence a name. We can then execute this sequence at any time by referring to the name.

Functions, Informally

- The part of the program that creates a function is called a *function definition*.
- When the function is used in a program, we say the definition is *called* or *invoked*.

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

Program Modules in C

- 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

Parameters and arguments

- Inside the function, the values that are passed get assigned to variable called **parameters**
  - `void sing(person)`

- The values that control how the function does its job called arguments (real values)
  - `Sing(“Bob”)`

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

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

Functions

- 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

Function Definitions

- Function definition format

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

  - Function-name: any valid identifier
  - Return-value-type: data type of the result (default `int`)
    - `void` - function returns nothing
  - Parameter-list: comma separated list, declares parameters (default `int`)

Function Definitions (II)

```
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;`*

Examples

```
/* getline:  get line into s, return length */
int getline(char s[], int lim)
{
    int c, i;

    i = 0;
    while (--lim > 0 && (c=getchar()) != EOF && c != '\n')
        s[i++] = c;
    if (c == '\n')
        s[i++] = c;
    s[i] = '\0';
    return i;
}
```

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 **`int`**s
    - Returns an **`int`**
- Promotion rules and conversions
  - Converting to lower types can lead to errors

Calling Functions: Call by Value

- 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

- Call by reference (*)
  - Passes original argument
  - Changes in function effect original
  - Only used with trusted functions

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

Recursion

- Example: factorial:

```
5! = 5 * 4 * 3 * 2 * 1
```

Notice that

```
5! = 5 * 4!
4! = 4 * 3! ...
```

  - Can compute factorials recursively
  - Solve base case (1! = 0! = 1) then plug in
    - `2! = 2 * 1! = 2 * 1 = 2;`
    - `3! = 3 * 2! = 3 * 2 = 6;`

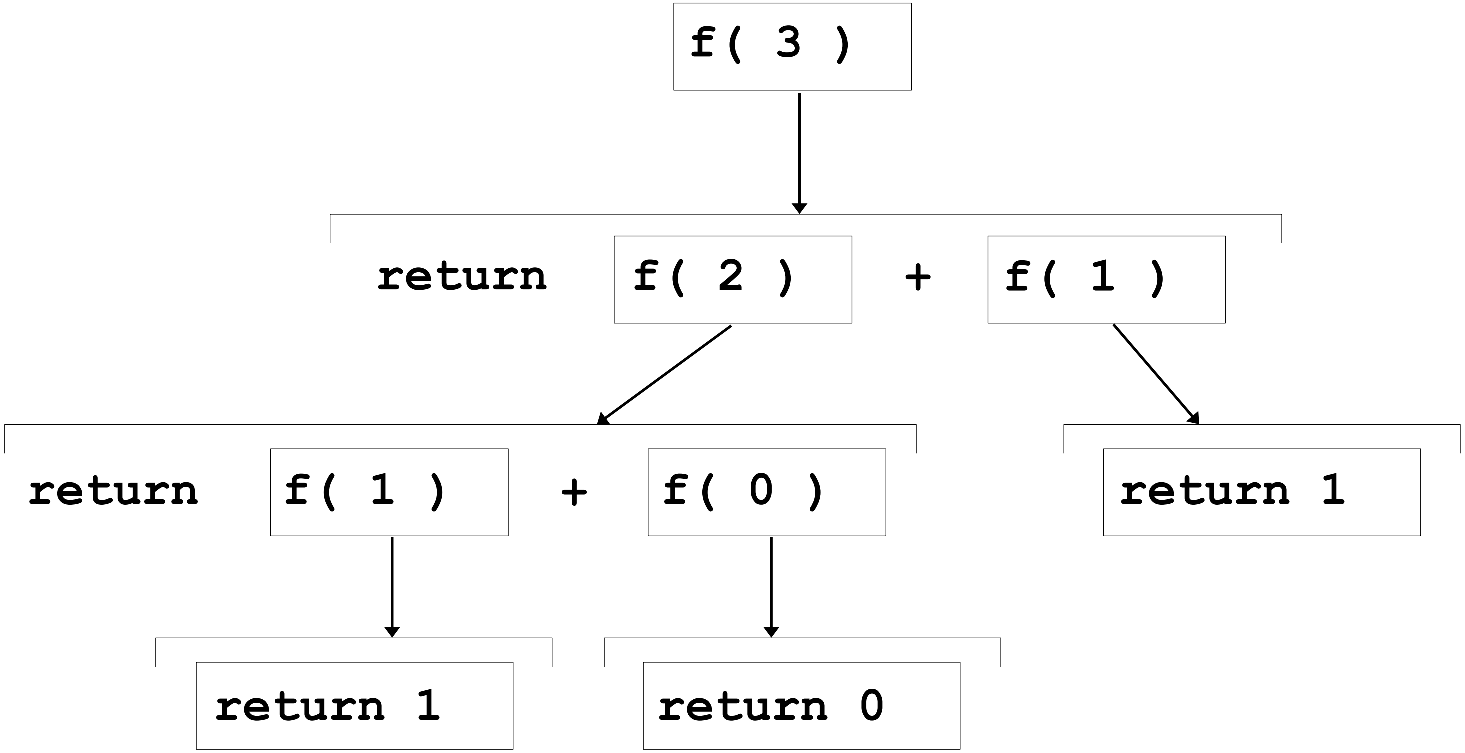
The Fibonacci Sequence

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

    `fib(n) = fib(n-1) + fib(n-2)` - recursive formula

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

The Fibonacci Sequence



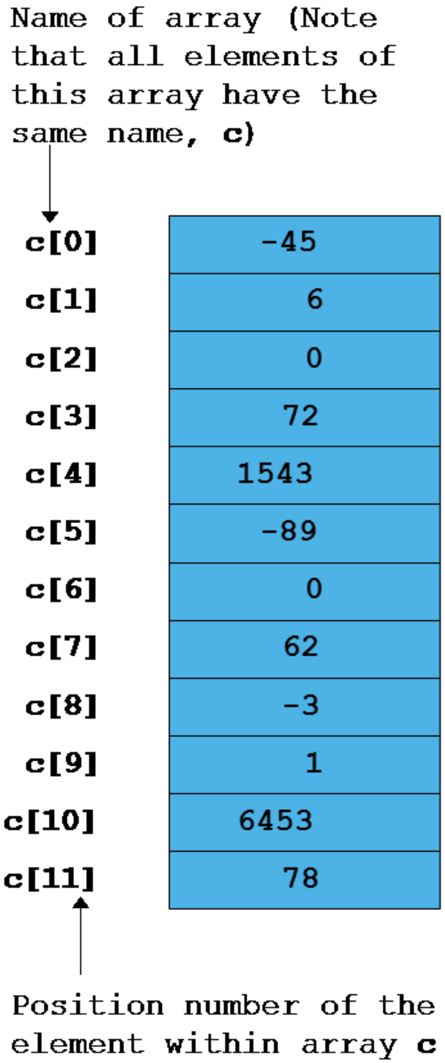
```c
/* Fig. 5.15: fig05_15.c
   Recursive fibonacci function */
#include <stdio.h>

long fibonacci( long );

int main()
{
   long result, number;

   printf( "Enter an integer: " );
   scanf( "%ld", &number );
   result = fibonacci( number );
   printf( "Fibonacci( %ld ) = %ld\n", number, result );
   return 0;
}

/* Recursive definition of function fibonacci */
long fibonacci( long n )
{
   if ( n == 0 || n == 1 )
      return n;
   else
      return fibonacci( n - 1 ) + fibonacci( n - 2 );
}
```

```
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
```

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]`



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**

Arrays

- Array elements are like normal variables

```
c[0] =  3;
printf( "%d", c[0] );
```

  - Perform operations in subscript. If `x = 3`,

```
c[5-2] == c[3] == c[x]
```

Declaring Arrays

- When declaring arrays, specify
  - Name
  - Type of array
  - Number of elements

```
arrayType arrayName[ numberOfElements ];
int c[ 10 ];
float myArray[ 3284 ];
```

- Declaring multiple arrays of same type
  - Format similar to regular variables

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

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}
```

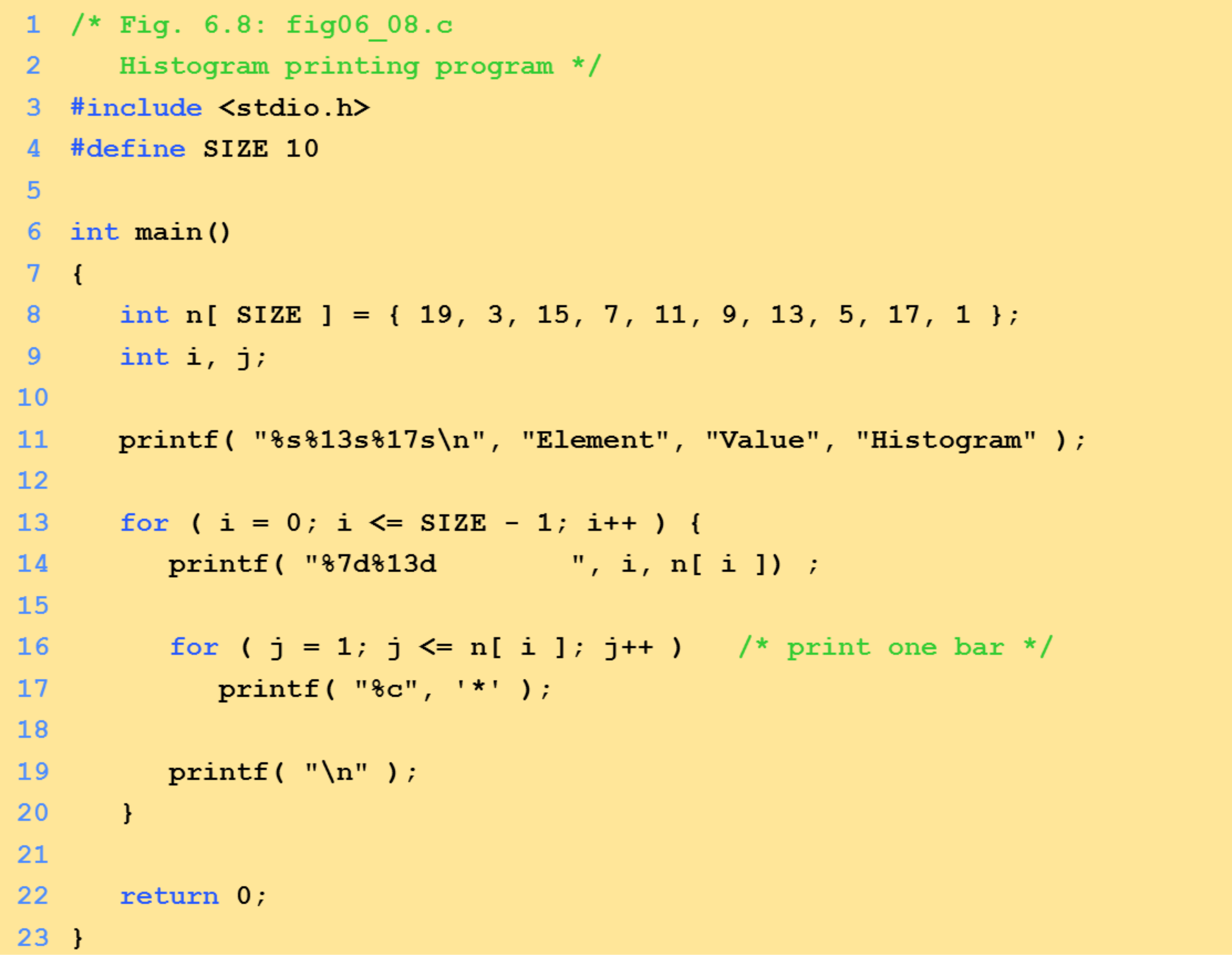
    - All elements 0

  - 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
/* Fig. 6.8: fig06_08.c
   Histogram printing program */
#include <stdio.h>
#define SIZE 10

int main()
{
   int n[ SIZE ] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };
   int i, j;

   printf( "%s%13s%17s\n", "Element", "Value", "Histogram" );

   for ( i = 0; i <= SIZE - 1; i++ ) {
      printf( "%7d%13d        ", i, n[ i ]) ;

      for ( j = 1; j <= n[ i ]; j++ )   /* print one bar */
         printf( "%c", '*' );

      printf( "\n" );
   }

   return 0;
}
```

- 1. Initialize array
- 2. Loop
- 3. Print

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

- Program Output

Examples

- 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' };
```

Examples

- 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"
   int i;

   printf(" Enter a string: ");
   scanf( "%s", string1 );
   printf( "string1 is: %s\nstring2: is %s\n"
           "string1 with spaces:\n",
           string1, string2 );

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

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

- 1. Initialize strings
- 2. Print strings
- 2.1 Define loop
- 2.2 Print characters
- individually
- 2.3 Input string
- 3. Print string

```
Enter a string: Hello there
string1 is: Hello
string2 is: string
string1 with spaces:
H e l l o
```

- Program Output

Passing Arrays to Functions

- Passing arrays
  - Specify array name without brackets

    ```
    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 Arrays to Functions

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

```
void modifyArray( int b[], int arraySize );
```

  - – Parameter names optional in prototype
    - **`int b[]`** could be simply **`int []`**
    - **`int arraySize`** could be simply **`int`**

```c
/* Fig. 6.13: fig06 13.c
   Passing arrays and individual array elements to functions */
#include <stdio.h>
#define SIZE 5

void modifyArray( int [], int );  /* appears strange */
void modifyElement( int );

int main()
{
   int a[ SIZE ] = { 0, 1, 2, 3, 4 }, i;

   printf( "Effects of passing entire array call "
           "by reference:\n\nThe values of the "
           "original array are:\n" );

   for ( i = 0; i <= SIZE - 1; i++ )
      printf( "%3d", a[ i ] );

   printf( "\n" );
   modifyArray( a, SIZE );  /* passed call by reference */
   printf( "The values of the modified array are:\n" );

   for ( i = 0; i <= SIZE - 1; i++ )
      printf( "%3d", a[ i ] );

   printf( "\n\n\nEffects of passing array element call "
           "by value:\n\nThe value of a[3] is %d\n", a[ 3 ] );
   modifyElement( a[ 3 ] );
   printf( "The value of a[ 3 ] is %d\n", a[ 3 ] );
   return 0;
}
```

- 1. Function definitions
- 2. Pass array to a function
- 2.1 Pass array o a function
- 3. Print

Entire arrays passed call-by-reference, and can be modified

Array elements passed call-by-value, and cannot be modified

```c

void modifyArray( int b[], int size )
{
   int j;

   for ( j = 0; j <= size - 1; j++ )
      b[ j ] *= 2;
}

void modifyElement( int e )
{
   printf( "Value in modifyElement is %d\n", e
}
```

- 3.1 Function definitions

```
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
```

- Program Output

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

```c
/* Fig. 6.16: fig06_16.c
   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, 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 };

   mean( response );
   median( response );
   mode( frequency, response );
   return 0;
}
```

- 1. Function prototypes
- 1.1 Initialize array
- 2. Call functions `mean`, `median`, and `mode`

```c

void mean( const int answer[] )
{
   int j, total = 0;

   printf( "%s\n%s\n%s\n", "********", "  Mean", "********" );

   for ( j = 0; j <= SIZE - 1; j++ )
      total += answer[  j  ];

   printf( "The mean is the average value of the data\n"
           "items. The mean is equal to the total of\n"
           "all the data items divided by the number\n"
           "of data items ( %d ). The mean value for\n"
           "this run is: %d / %d = %.4f\n\n",
           SIZE, total, SIZE, (  double  ) total / SIZE );
}

void median( int answer[] )
{
   printf( "\n%s\n%s\n%s\n%s",
           "********", " Median", "********",
           "The unsorted array of responses is" );

   printArray( answer );
   bubbleSort( answer );
   printf( "\n\nThe sorted array is" );
   printArray( answer );
   printf( "\n\nThe median is element %d of\n"
           "the sorted %d element array.\n"
           "For this run the median is %d\n\n",
           SIZE / 2, SIZE, answer[ SIZE / 2 ] );
```

- 3. Define function `mean`
- 3.1 Define function `median`
- 3.1.1 Sort Array
- 3.1.2 Print middle element

```c
}

void mode( int freq[], const int answer[] )
{
   int rating, j, h, largest = 0, modeValue = 0;

   printf( "\n%s\n%s\n%s\n",
           "********", "  Mode", "********" );

   for ( rating = 1; rating <= 9; rating++ )
      freq[ rating ] = 0;

   for ( j = 0; j <= SIZE - 1; j++ )
      ++freq[ answer[ j ] ];

   printf( "%s%11s%19s\n\n%54s\n%54s\n\n",
           "Response", "Frequency", "Histogram",
           "1    1    2    2", "5    0    5    0    5" );

   for ( rating = 1; rating <= 9; rating++ ) {
      printf( "%8d%11d          ", rating, freq[ rating ] );

      if ( freq[ rating ] > largest ) {
         largest = freq[ rating ];
         modeValue = rating;
      }

      for ( h = 1; h <= freq[ rating ]; h++ )
         printf( "*" );

```

Notice how the subscript in **`frequency[]`** is the value of an element in **`response[]`** (**`answer[]`**)

Print stars depending on value of **`frequency[]`**

3.2 Define function `mode`

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

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

   printf( "The mode is the most frequent value.\n"
           "For this run the mode is %d which occurred"
           " %d times.\n", modeValue, largest );
}

void bubbleSort( int a[] )
{
   int pass, j, hold;

   for ( pass = 1; pass <= SIZE - 1; pass++ )

      for ( j = 0; j <= SIZE - 2; j++ )

         if ( a[ j ] > a[ j + 1 ] ) {
            hold = a[ j ];
            a[ j ] = a[ j + 1 ];
            a[ j + 1 ] = hold;
         }
}

void printArray( const int a[] )
{
   int j;

   for ( j = 0; j <= SIZE - 1; j++ ) {

      if ( j % 20 == 0 )
         printf( "\n" );
```

Bubble sort: if elements out of order, swap them.

- 3.3 Define `bubbleSort`
- 3.3 Define `printArray`

```

      printf( "%2d", a[ j ] );
   }
}
```

```
********
 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 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

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

- Program Output

```
********
  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.
```

- Program Output

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

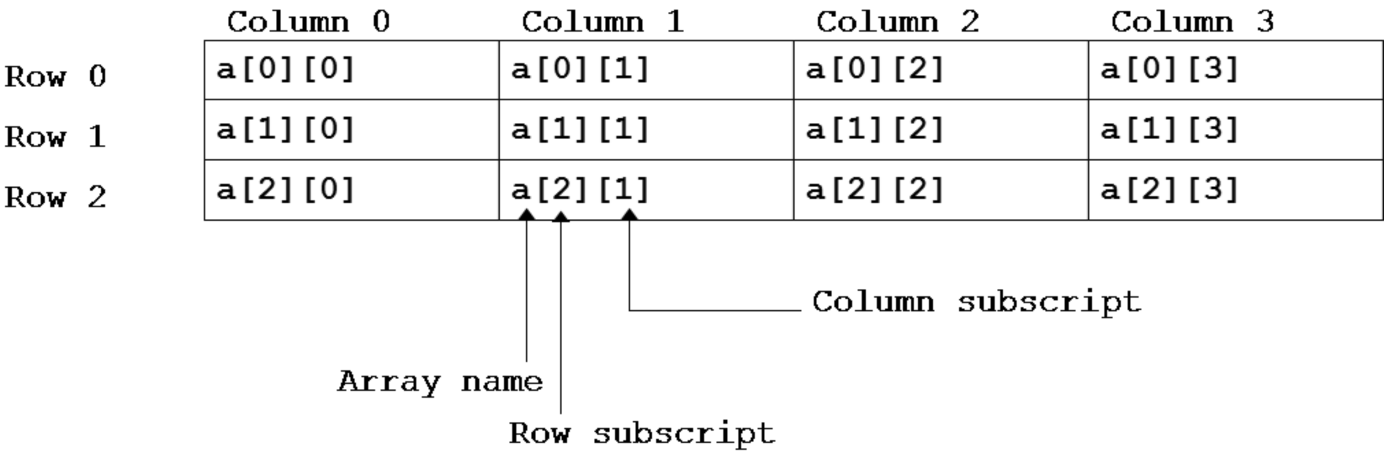
Searching Arrays: Linear Search and Binary Search (II)

- 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

      $2^5 > 30$

Multiple-Subscripted Arrays

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



	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

Multiple-Subscripted Arrays

- Initialization

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

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 } };
```

1	0
3	4

- Referencing elements

  – Specify row, then column

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

Program structure

- C Preprocessor
- Storage Class
- Scope Rules

File inclusion

- Include library files (#include <stdio.h>)
- Include user source files (#include "sqrt.c")
  - Need to be put in the same folder
  - In the same project (CodeBlocks)
- Include user pre-compile files (#include "sqrt.h")
  - Need to be put in the same folder
  - In the same project (CodeBlocks)

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

Macro Substitution

- Define constants (#define MON_MAX 100)
- Replacement of texts
  - Simple: #define BEGIN {
  - Yet: #define forever for (;;)
  - With arguments (be very careful!!!): #define max(A,B) ((A)>(B)?(A):(B))
  - Multiple lines: with \ at the end of the line

Example

```
#include <stdio.h>
#include <limits.h>
#define MAX(a,b)  (a>=b)?a:b

int main()
{
    int i=1,j=2,k;
    k=MAX(i++, j++);
    printf("i=%d, j=%d and k=%d\n", i, j, k);
    return 0;
}
```

Conditional inclusion

- Control preprocessing itself with conditional statements
- Include code selectively
- #ifndef, #if, #elseif, #else
- Eg:

```
#if (INT_MAX==2^31-1)
    printf("32bits machine \n");
#else
    printf("other machine \n");
#endif
```

Storage Classes

- Storage class specifies
  - Scope - where object can be referenced in program
  - Storage duration - how long an object exists in memory
  - Linkage - what files an identifier is known

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.)

Scope Rules

- 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

Storage Classes

- 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 (incompatible with `static`)
    - `register int counter = 1;`

Storage Classes

- 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 (for different calls of functions)
    - `Static functionName` makes it invisible outside the file

Static variable

```
# include <stdio.h>
void augmente ( void )
{
  auto int i =0;
  printf("i=%d \ n", i++);
}

int main ( )
{
  auto int j;
  for(j=1; j<=3; j++)
    augmente();
  return (0);
}
```

```
# include <stdio.h>
void augmente ( void )
{
  static int i =0;
  printf("i=%d \ n", i++);
}

int main ( )
{
  auto int j;
  for(j=1; j<=3; j++)
    augmente();
  return (0);
}
```

External Variable

- `extern:` global variables and functions.
  - Known in any function (shared value)
  - A declaration not a definition (no memory allocation)
  - Definition somewhere else, and permanent!
  - Very useful in header files (.h)

//File essai.h

```
#ifndef ESSAI_H
#define ESSAI_H
extern int i;
extern int j;
extern void augmente (void);
#endif
```

Protection

- Use **static** with **extern**

```
//File essai.c
#include <stdio.h>
int i;
static int j;

void augmente ( )
{
  printf("i=%d, j=%d \n",
i++, j++);
}
```

```
//File main.c
#include <stdio.h>
#include "essai.h"

int main()
{
  int k;
  for (k=1; k<=3; k++)
         augmente ( ) ;
   printf(“Finally i=%d,
j=%d\n", i, j);
   return (0);
}
```