#include<stdlib.h>

This header file defines several general purpose functions, including dynamics memory management, random number generation, communication with the environment, integer arithmetic, searching, sorting and converting.

Pseudo-random sequence generation

rand	Generate random number (function)
srand	Initialize random number generator (function)

int rand (void); void srand (unsigned int seed);

#include<stdlib.h>

String conversion

atof	Convert string to double (function)
atoi	Convert string to integer (function)
atol	Convert string to long integer (function)
atoll	Convert string to long long integer (function)
strtod	Convert string to double (function)
strtof	Convert string to float (function)
strtol	Convert string to long integer (function)
strtold	Convert string to long double (function)
strtoll	Convert string to long long integer (function)
strtoul	Convert string to unsigned long integer (function)
strtoull	Convert string to unsigned long long integer (function)

#include<stdlib.h>

Dynamic memory management

calloc	Allocate and zero-initialize array (function)
free	Deallocate memory block (function)
malloc	Allocate memory block (function)
realloc	Reallocate memory block (function)

Integer arithmetics

abs	Absolute value (function)
div	Integral division (function)
labs	Absolute value (function)
ldiv	Integral division (function)
llabs	Absolute value (function)
lldiv	Integral division (function)

#include<stdlib.h>

Environment

abort	Abort current process (function)
atexit	Set function to be executed on exit (function)
at_quick_exit	Set function to be executed on quick exit (function)
exit	Terminate calling process (function)
getenv	Get environment string (function)
quick_exit	Terminate calling process quick (function)
system	Execute system command (function)
_Exit	Terminate calling process (function)

Searching and sorting

bsearch	Binary search in array (function)
qsort	Sort elements of array (function)

Random Number Generation

- **rand** function
  - Load **<stdlib.h>**
  - Returns "random" number between **0** and **RAND_MAX** (at least **32767**)

```
i = rand();
```

  - Pseudorandom
    - Preset sequence of "random" numbers
    - Same sequence for every function call
- Scaling
  - To get a random number between **1** and **n**

```
1 + ( rand() % n )
```

    - **rand() % n** returns a number between **0** and **n - 1**
    - Add **1** to make random number between **1** and **n**

```
1 + ( rand() % 6)
```

      - number between **1** and **6**

Random Number Generation

- **srand** function
  - **<stdlib.h>**
  - Takes an integer seed and jumps to that location in its "random" sequence

```
srand( seed );
```

  - **srand(time(NULL)); //load <time.h>**
    - **time(NULL)**
      - Returns the time at which the program was compiled in seconds
      - "Randomizes" the seed

```
/*Randomizing die-rolling program */

#include<stdlib.h>
#include<stdio.h>
 Int main()
 {
   int  i;
   unsigned  seed;
   printf( "Enter seed: " );
   scanf( "%u", &seed );
   srand( seed );
   for( i = 1; i <= 10; i++ )
     {
          printf( "%10d", 1 + ( rand() % 6 ) );
          if( i % 5 == 0 )
               printf( "\n" );
          }
   return 0;
  }
```

Program Output

```
Enter seed: 67
 6  1  4  6  2
 1  6  1  6  4

Enter seed: 867
 2   4   6   1   6
1   1   3   6   2

Enter seed: 67
6  1  4  6  2
1  6  1  6  4
```

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

```
/* Craps */
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
Int  rollDice( void);
Int  main()
  {
    int  gameStatus, sum, myPoint;
    srand( time( NULL ) );
    sum = rollDice();              /* first roll of the dice */
    switch ( sum ) {
        case7: case11:             /* win on first roll */
            gameStatus = 1;
        break;
        case2: case3: case12:      /* lose on first roll */
            gameStatus = 2;
        break;
        default:                   /* remember point */
            gameStatus = 0;
            myPoint = sum;
            printf( "Point is %d\n", myPoint );
        break;
    }
```

```
while( gameStatus == 0 )
   {        /* keep rolling */
     sum = rollDice();
     if( sum == myPoint )       /* win by making point */
            gameStatus = 1;
    else
            if( sum == 7 )      /* lose by rolling 7 */
               gameStatus = 2;
      }
 if( gameStatus == 1 )
            printf( "Player wins\n" );
 else
            printf( "Player loses\n" );
 return0;
}

Int  rollDice( void)
{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
Player rolled 6 + 6 = 12
Player loses

Player rolled 4 + 6 = 10
Point is 10
Player rolled 2 + 4 = 6
Player rolled 6 + 5 = 11
Player rolled 3 + 3 = 6
Player rolled 6 + 4 = 10
Player wins

Player rolled 1 + 3 = 4
Point is 4
Player rolled 1 + 4 = 5
Player rolled 5 + 4 = 9
Player rolled 4 + 6 = 10
Player rolled 6 + 3 = 9
Player rolled 1 + 2 = 3
Player rolled 5 + 2 = 7
Player loses
```

Storage Classes

- Storage class specifiers
  - Storage duration – how long an object exists in memory
  - Scope – where object can be referenced in program
  - Linkage – specifies the files in which an identifier is known
- 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;
```

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
  - **`extern`**: default for global variables and functions
    - Known in any function

Scope Rules

- File scope
  - Identifier defined outside function, known in all functions
  - Used for global variables, function definitions, function prototypes
- Function scope
  - Can only be referenced inside a function body
  - Used only for labels (`start:`, `case:` , etc.)

Scope Rules

- Block scope
  - Identifier declared inside a block
    - Block scope begins at declaration, ends at right brace
  - Used for variables, function parameters (local variables of function)
  - Outer blocks "hidden" from inner blocks if there is a variable with the same name in the inner block
- Function prototype scope
  - Used for identifiers in parameter list

```
/* scoping example */
#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 );
{
/* start new scope */
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 */
```

```
printf( "local x in main is %d\n", x );
return0;
 }
Void a( void)
{int x = 25;            /* initialized each time a is called */
 printf( "\nlocal x in a is %d after entering a\n", x );
  ++x;
  printf( "local x in a is %d before exiting a\n", x );
  }

Void b( void)
{
static int x = 50;      /* static initialization only */
                        /* first time b is called */
printf( "\nlocal static x is %d on entering b\n", x );
++x;
printf( "local static x is %d on exiting b\n", x );
}
Void c( void)
{
printf( "\nglobal x is %d on entering c\n", x );
x *= 10;
printf( "global x is %d on exiting c\n", x );
}
```

Program Output

```
local x in outer scope of main is 5
local x in inner scope of main is 7
local x in outer scope of main is 5

local x in a is 25 after entering a
local x in a is 26 before exiting a

local static x is 50 on entering b
local static x is 51 on exiting b

global x is 1 on entering c
global x is 10 on exiting c

local x in a is 25 after entering a
local x in a is 26 before exiting a

local static x is 51 on entering b
local static x is 52 on exiting b

global x is 10 on entering c
global x is 100 on exiting c
local x in main is 5
```

Recursion

- Recursive functions
  - Functions that call themselves
  - Can only solve a base case
  - Divide a problem up into
    - What it can do
    - What it cannot do
      - What it cannot do resembles original problem
      - The function launches a new copy of itself (recursion step) to solve what it cannot do
  - Eventually base case gets solved
    - Gets plugged in, works its way up and solves whole problem

Recursion

- Example: factorials
  - `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;`

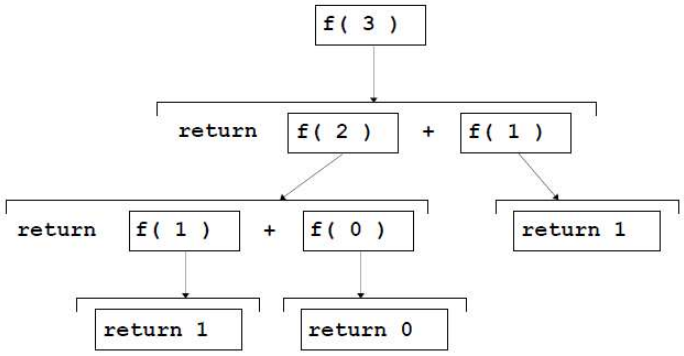
Example Using Recursion: The Fibonacci Series

- Fibonacci series: 0, 1, 1, 2, 3, 5, 8...
  - Each number is the sum of the previous two
  - Can be solved recursively:
    - `fib( n ) = fib( n - 1 ) + fib( n - 2 )`
  - Code for the `fibaonacci` function

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

Example Using Recursion: The Fibonacci Series

- Set of recursive calls to function **fibonacci**




```
/* 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 );
return0;
}
/* Recursive definition of function fibonacci */
Long fibonacci ( long n ) {
 if( n == 0 || n == 1 )
  return n;
 else
  return fibonacci( n -1 ) + fibonacci( n -2 );
}
```

OUTPUT

```
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
```

OUTPUT

```
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
```

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)

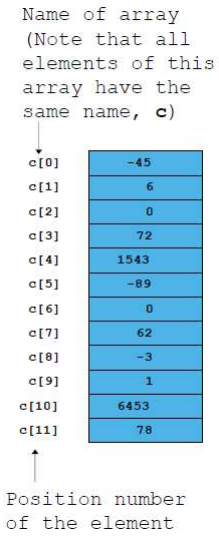
ARRAYS

- Arrays
  - Structures of related data items
  - Static entity – same size throughout program
  - Dynamic data structures

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]



ARRAYS

- Array elements are like normal variables

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

  - Perform operations in subscript. If **x** equals 3

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

Declaring ARRAYS

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

```
arrayType arrayName[ numberOfElements ];
```

  - Examples:

```
int c[ 10 ];
float myArray[ 3284 ];
```

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

```
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

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

    - All elements 0
  - If too many a syntax error is produced syntax error
  - 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

```
/* Histogram printing program */
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;
}
```

Output

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

Examples Using Arrays

- Character arrays
  - String "`first`" 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
      - It is equivalent to

```
char string1[] = { 'f', 'i', 'r', 's', 't', '\0' };
```

  - Can 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

```
/* 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\nstring2: is %s\n",string1,string2);
Printf("string1 with spaces between characters is:\n" );
for( i = 0; string1[ i ] != '\0'; i++ )
            printf( "%c ", string1[ i ] );
printf( "\n" );
return0;
}
```

OUTPUT On Execution:

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

Passing Arrays to Functions

- Passing arrays
  - To pass an array argument to a function, specify the name of the array without any 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 array elements
  - Passed by call-by-value
  - Pass subscripted name (i.e., `myArray[ 3 ]`) to function

Passing Arrays to Functions

- Function prototype

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

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

```
/* 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 ] );
return0;
}
```

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

Entire Arrays passed call-by Value , and cannot be modified

```
Void modifyArray( intb[], intsize )
{int j;
for( j = 0; j <= size -1; j++ )
b[ j ] *= 2;
}

Void modifyElement( int e )
{
printf( "Value in modifyElement is %d\n", e *= 2 );
}
```

Output on Execution:

```
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
```

Sorting Arrays

- Sorting data
  - Important computing application
  - Virtually every organization must sort some data
- 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

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, 9,1,8,7, 8, 9, 5, 9, 8, 7, 8, 7, 8,1,9,6, 7, 8, 9, 3, 9, 8, 7, 8,
7,2,7, 8, 9, 8, 9, 8, 9, 7, 8, 9,2,1,6, 7, 8, 7, 8, 7, 9, 8, 9, 2,2,2,7, 8, 9, 8, 9, 8, 9, 7, 5,
3,2,3,5, 6, 7, 2, 5, 3, 9, 4, 6, 4,2,4,7, 8, 9, 6, 8, 7, 8, 9, 7, 8,2,5,7, 4, 4, 2, 5, 3, 8, 7,
5, 6,2,6,4, 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( "%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 ] );
}
```

```
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( "*" );

        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 );
}
```

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

Print stars depending on the values in freq[]

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

Program OUTPUT

```
********
  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 forth
is 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 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 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	*******************

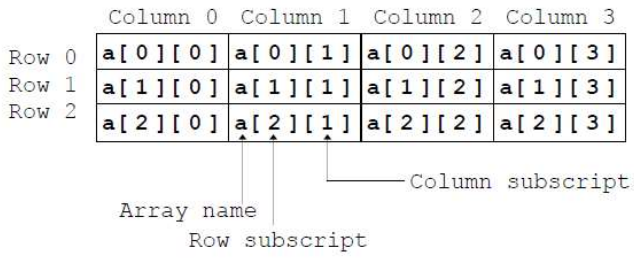
The mode is the most frequent value.
For this run the mode is 8 which occurred 27 times.

Searching Arrays: Linear Search and Binary Search

- 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
      - $2^5 > 30$ so at most 5 steps

Multiple-Subscripted Arrays

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



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 ] );
```

```
/* Double-subscripted array example */
#include<stdio.h>
#define STUDENTS
#define EXAMS
int  minimum( const int [][ EXAMS ], int, int);
int maximum( const int [][ EXAMS ], int, int);
double average( const int [], int);
Void printArray( const int [][ EXAMS ], 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( "\nstudentGrades[%d] ", i );
for( j = 0; j <= tests -1; j++ )
        printf( "%-5d", grades[ i ][ j ] );
   }
}
```

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