Functions

Math Library Functions

- Perform common mathematical calculations.
- Include the header file **`<cmath>`**
- Functions called by writing : `functionName (argument)`;or `functionName(argument1, argument2,…);`
- Example:

```
cout << sqrt(900.0);
```

sqrt (square root) function. The preceding statement would print 30.

- All functions in math library return a **`double`**.
- Function arguments can be:
  - Constants **`sqrt(4);`**
  - Variables **`sqrt(x);`**
  - Expressions **`sqrt(sqrt(x)); sqrt(3 - 6x);`**

Math Library Functions

Method	Description	Example

ceil(x)	rounds *x* to the smallest integer not less than *x*	`ceil( 9.2 )` is `10.0`
		`ceil( -9.8 )` is `-9.0`

cos(x)	trigonometric cosine of *x* (*x* in radians)	`cos( 0.0 )` is `1.0`

exp(x)	exponential function *ex*	`exp( 1.0 )` is `2.71828`
		`exp( 2.0 )` is `7.38906`

fabs(x)	absolute value of *x*	`fabs( 5.1 )` is `5.1`
		`fabs( 0.0 )` is `0.0`
		`fabs( -8.76 )` is `8.76`

floor(x)	rounds *x* to the largest integer not greater than *x*	`floor( 9.2 )` is `9.0`
		`floor( -9.8 )` is `-10.0`

fmod(x, y)	remainder of *x/y* as a floating-point number	`fmod( 13.657, 2.333 )` is `1.992`

log(x)	natural logarithm of *x* (base *e*)	`log( 2.718282 )` is `1.0`
		`log( 7.389056 )` is `2.0`

log10(x)	logarithm of *x* (base 10)	`log10( 10.0 )` is `1.0`
		`log10( 100.0 )` is `2.0`

pow(x, y)	*x* raised to power *y* (*xy*)	`pow( 2, 7 )` is `128`
		`pow( 9, .5 )` is `3`

sin(x)	trigonometric sine of *x* (*x* in radians)	`sin( 0.0 )` is `0`

sqrt(x)	square root of *x*	`sqrt( 900.0 )` is `30.0`
		`sqrt( 9.0 )` is `3.0`

tan(x)	trigonometric tangent of *x* (*x* in radians)	`tan( 0.0 )` is `0`

Fig. 3.2 Math library functions.

Functions

➢Functions :

- Modularize a program which allows for software re-usability
- Call function multiple times
- Local variables
- Known only in the function in which they are defined
- All variables declared in function definitions are local variables
- Parameters
- Local variables passed to function when called
- Provide outside information

Function Definitions

- Function prototype
- Tells compiler argument type and return type of function

  `int square( int );`

  Function takes an **`int`** and returns an **`int.`**

  - Explained in more detail later

- Calling/invoking a function: **`square(x);`**
  - Parentheses an operator used to call function
  - Pass argument x
  - Function gets its own copy of arguments
  - After finished, passes back result

Function Definitions

- Format for function definition

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

- Parameter list, Comma separated list of arguments
- Data type needed for each argument
- If no arguments, use **`void`** or leave blank
- Return-value-type
- Data type of result returned (use **`void`** if nothing returned)

Function Definitions

- Example function:

```
int square( int y )
{
 return y * y;
}
```

- **`return`** keyword: Returns data, and control goes to function’s caller. If no data to return, use **`return;`**
- Function ends when reaches right brace
- Control goes to caller.
- Functions cannot be defined inside other functions

Square function

```cpp
 // Fig. 3.3: fig03_03.cpp
 // Creating and using a programmer-defined function.
 #include <iostream>

 using std::cout;
 using std::endl;

 int square( int );   // function prototype

 int main()
 {
   // loop 10 times and calculate and output
   // square of x each time
   for ( int x = 1; x <= 10; x++ )
      cout << square( x ) << "  ";  // function call

   cout << endl;

   return 0;  // indicates successful termination

 } // end main

// square function definition returns square of an integer
int square( int y )  // y is a copy of argument to function
{
   return y * y;     // returns square of y as an int

} // end function square
```

squareFunc.cpp

Square function

```cpp
 // Fig. 3.3: fig03_03.cpp
 // Creating and using a programmer-defined function.
 #include <iostream>

 using std::cout;
 using std::endl;

 int square( int );   // function prototype

 int main()
 {
   // loop 10 times and calculate and output
   // square of x each time
   for ( int x = 1; x <= 10; x++ )
      cout << square( x ) << "  ";  // function call

   cout << endl;

   return 0;  // indicates successful termination

 } // end main

// square function definition returns square of an integer
int square( int y )  // y is a copy of argument to function
{
   return y * y;     // returns square of y as an int

} // end function square
```

Parentheses **()** cause function to be called. When done, it returns the result.

squareFunc.cpp

Square function

```cpp
 // Fig. 3.3: fig03_03.cpp
 // Creating and using a programmer-defined function.
 #include <iostream>

 using std::cout;
 using std::endl;

 int square( int );   // function prototype

 int main()
 {
   // loop 10 times and calculate and output
   // square of x each time
   for ( int x = 1; x <= 10; x++ )
      cout << square( x ) << "  ";  // function call

   cout << endl;

   return 0;  // indicates successful termination

 } // end main

// square function definition returns square of an integer
int square( int y )  // y is a copy of argument to function
{
   return y * y;     // returns square o

} // end function square
```

Definition of **`square`**. y is a copy of the argument passed. Returns y*y, or y squared.

squareFunc.cpp

Square function

```cpp
 // Fig. 3.3: fig03_03.cpp
 // Creating and using a programmer-defined function.
 #include <iostream>

 using std::cout;
 using std::endl;

 int square( int );   // function prototype

 int main()
 {
   // loop 10 times and calculate and output
   // square of x each time
   for ( int x = 1; x <= 10; x++ )
      cout << square( x ) << "  ";  // function call

   cout << endl;

   return 0;  // indicates successful termination

 } // end main

// square function definition returns square of an integer
int square( int y )  // y is a copy of argument to function
{
   return y * y;     // returns square of y as an int

} // end function square
```

```
1  4  9  16  25  36  49  64  81  100
```

squareFunc.cpp

Maximum function

```cpp
// Fig. 3.4: fig03_04.cpp
// Finding the maximum of three floating-point numbers.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

double maximum( double, double, double ); // function prototype

int main()
{
  double number1;
  double number2;
  double number3;

  cout << "Enter three floating-point numbers: ";
  cin >> number1 >> number2 >> number3;

  // number1, number2 and number3 are arguments to
  // the maximum function call
  cout << "Maximum is: "
       << maximum( number1, number2, number3 ) << endl;

  return 0;  // indicates successful termination
```

Function **`maximum`** takes 3 arguments (all **`double`**) and returns a **`double`**.

maximumFunc.cpp

Maximum function

```

} // end main

// function maximum definition;
// x, y and z are parameters
double maximum( double x, double y, double z )
{
   double max = x;    // assume x is largest

   if ( y > max )     // if y is larger,
      max = y;        // assign y to max

   if ( z > max )     // if z is larger,
      max = z;        // assign z to max

   return max;        // max is largest value

} // end function maximum
```

Comma separated list for multiple parameters

```
Enter three floating-point numbers: 99.32 37.3 27.1928
Maximum is: 99.32

Enter three floating-point numbers: 1.1 3.333 2.22
Maximum is: 3.333

Enter three floating-point numbers: 27.9 14.31 88.99
Maximum is: 88.99
```

maximumFunc.cpp

Header Files

- Header files contain
  - Function prototypes
  - Definitions of data types and constants
- Header files ending with .h
  - Programmer-defined header files

    ```
    #include “myheader.h”
    ```
- Library header files

    ```
    #include <cmath>
    ```

Function definition in an external macro

Example: Create a library of function to convert temperature from degrees Celsius to Fahrenheit, from Celsius to Kelvin from Fahrenheit to Kelvin and vice-versa

Functions declaration must be in a .h file and implemented in a .cpp file. Create also a main program that prints the options, ask the temperature to be converted, makes the conversion and prints the results.

```
./temperatureConverter.exe
To convert arbitrary temperatures, enter:
   A- to convert Fahrenheit to Celsius;
   B- to convert Celsius to Fahrenheit;
   C- to convert Celsius to Kelvin;
   D- to convert Kelvin to Celsius;
   E- to convert Fahrenheit to Kelvin; or
   F- to convert Kelvin to Fahrenheit.
--> c
Enter the temperature to be converted: 45
The converted temperature is 318.15
```

Temperature Converter

```
// temperature.h
// declare functions to convert temperatures

double FahrToCelsius(double);      // convert from Fahrenheit to Celsius

double CelsiusToFahr(double);      // convert from Celsius to Fahrenheit

double FahrToKelvin(double);       // convert from Fahrenheit to Kelvin

double KelvinToFahr(double);       // convert from Kelvin to Fahrenheit

double CelsiusToKelvin(double);    // convert from Celsius to Kelvin

double KelvinToCelsius(double);    // convert from Kelvin to Celsius
```

temperature.h

Temperature Converter

```
// temperature.cpp
// definizione funzioni per conversioni
temperature
#include "temperature.h"
//--------------------------------------------
double FahrToCelsius(double tempFahr)
{
  return (tempFahr - 32.0) / 1.8;
}
//--------------------------------------------
double CelsiusToFahr(double tempCels)
{
   return tempCels * 1.8 + 32.0;
}
//--------------------------------------------
double KelvinToCelsius(double tempKelv)
{
  return tempKelv - 273.15;
}
```

```
//--------------------------------------------
double CelsiusToKelvin(double tempCels)
{
  return tempCels + 273.15;
}
//--------------------------------------------
double FahrToKelvin(double tempFahr)
{
  return (tempFahr - 32.0) / 1.8 + 273.15;
}
//--------------------------------------------
double KelvinToFahr(double tempKelv)
{
  return (tempKelv - 273.15) * 1.8 + 32.0;
}
```

temperature.cpp

Temperature Converter

```
// Convert Temperature from one scale to the other

#include <iostream>
using std::cout;
using std::cin;
using std::endl;
using std::cerr;

#include <string>
using std::string;

#include "temperature.h"

int main()
{
   const string MENU = "To convert arbitrary temperatures, enter:\n"
                       "   A - to convert Fahrenheit to Celsius;\n"
                       "   B - to convert Celsius to Fahrenheit;\n"
                       "   C - to convert Celsius to Kelvin;\n"
                       "   D - to convert Kelvin to Celsius;\n"
                       "   E - to convert Fahrenheit to Kelvin; or\n"
                       "   F - to convert Kelvin to Fahrenheit.\n"
                       "--> ";
   cout << MENU;
   char conversion;
   cin >> conversion;
```

useTemperature.cpp

Temperature Converter

```
cout << "\nEnter the temperature to be converted: ";
double temperature;
cin >> temperature;

double result;

switch (conversion)
{
   case 'A': case 'a':
                        result = FahrToCelsius(temperature);
                        break;
   case 'B': case 'b':
                        result = CelsiusToFahr(temperature);
                        break;
   case 'C': case 'c':
                        result = CelsiusToKelvin(temperature);
                        break;
   case 'D': case 'd':
                        result = KelvinToCelsius(temperature);
                        break;
   case 'E': case 'e':
                        result = FahrToKelvin(temperature);
                        break;
```

useTemperature.cpp

Temperature Converter

```
      case 'F': case 'f':
                           result = KelvinToFahr(temperature);
                           break;
      default:
                           cerr << "\n*** Invalid conversion: "
                                << conversion << endl;
                           result = 0.0;
   }

   cout << "The converted temperature is " << result << endl;

   return 0;
}
```

How to compile your code in this case?

```
g++ temperature.cpp useTemperature.cpp -o temperature
```

useTemperature.cpp

Random Number Generation

- **`rand`** function (**`<cstdlib>`**)

  `i = rand();`

  - Generates unsigned integer between 0 and RAND_MAX (usually 32767)

- Scaling and shifting

  - Modulus (remainder) operator: `%`

    `10 % 3` is **`1`**

    `x % y` is between **`0`** and **`y – 1`**

  - Example

    `i = rand() % 6 + 1;`

    "**`Rand() % 6`**" generates a number between **`0`** and **`5`** (scaling)

    "**`+ 1`**" makes the range 1 to 6 (shift)

Roll dice

rolldice.cpp

```cpp
 // Fig. 3.7: fig03_07.cpp
 // Shifted, scaled integers produced by 1 + rand() % 6.
 #include <iostream>

 using std::cout;
 using std::endl;

 #include <iomanip>

using std::setw;

#include <cstdlib>   // contains function prototype for rand

int main()
{
   // loop 20 times
   for ( int counter = 1; counter <= 20; counter++ ) {

      // pick random number from 1 to 6 and output it
      cout << setw( 10 ) << ( 1 + rand() % 6 );

      // if counter divisible by 5, begin new line of output
      if ( counter % 5 == 0 )
         cout << endl;

   } // end for structure

   return 0;  // indicates successful termination

} // end main
```

Roll dice

rolldice.cpp

```
 // Fig. 3.7: fig03_07.cpp
 // Shifted, scaled integers produced by 1 + rand() % 6.
 #include <iostream>

 using std::cout;
 using std::endl;

 #include <iomanip>

using std::setw;

#include <cstdlib>   // contains function prototype for rand

int main()
{
   // loop 20 times
   for ( int counter = 1; counter <= 20; counter++ ) {

      // pick random number from 1 to 6 and output it
      cout << setw( 10 ) << ( 1 + rand() % 6 );

      // if counter divisible by 5, begin new line of output
      if ( counter % 5 == 0 )
         cout << endl;

   } // end for structure

   return 0;  // indicates successful termination

} // end main
```

Output of **`rand()`** scaled and shifted to be a number between 1 and 6.

Roll dice

rolldice.cpp

```
 // Fig. 3.7: fig03_07.cpp
 // Shifted, scaled integers produced by 1 + rand() % 6.
 #include <iostream>

 using std::cout;
 using std::endl;

 #include <iomanip>

using std::setw;

#include <cstdlib>   // contains function prototype for rand

int main()
{
   // loop 20 times
   for ( int counter = 1; counter <= 20; counter++ ) {

      // pick random number from 1 to 6 and output it
      cout << setw( 10 ) << ( 1 + rand() % 6 );

      // if counter divisible by 5, begin new line of output
      if ( counter % 5 == 0 )
         cout << endl;

   } // end for structure

   return 0;  // indicates successful termination

} // end main
```

```
         6         6         5         5         6
         5         1         1         5         3
         6         6         2         4         2
         6         2         3         4         1
```

Random Number Generation

- Calling rand() repeatedly, gives the same sequence of numbers
- Pseudorandom numbers:
  - Preset sequence of "random" numbers
  - Same sequence generated whenever program run
- To get different random sequences, we need to provide a seed value, which acts like a random starting point in the sequence
  - The same seed will give the same sequence

    `srand(seed);`

  - **`<cstdlib>`** Used before **`rand()`** to set the seed

Seed of the srand

```
 // Fig. 3.9: fig03_09.cpp
 // Randomizing die-rolling program.
 #include <iostream>

 using std::cout;
 using std::cin;
 using std::endl;

 #include <iomanip>

using std::setw;

// contains prototypes for functions srand and rand
#include <cstdlib>

// main function begins program execution
int main()
{
   unsigned seed;

   cout << "Enter seed: ";
   cin >> seed;
   srand( seed );  // seed random number generator

```

seed.cpp

Seed of the srand

```cpp
 // Fig. 3.9: fig03_09.cpp
 // Randomizing die-rolling program.
 #include <iostream>

 using std::cout;
 using std::cin;
 using std::endl;

 #include <iomanip>

using std::setw;

// contains prototypes for functions srand and rand
#include <cstdlib>

// main function begins program execution
int main()
{
   unsigned seed;

   cout << "Enter seed: ";
   cin >> seed;
   srand( seed );  // seed random number generator

```

Setting the seed with **`srand()`**

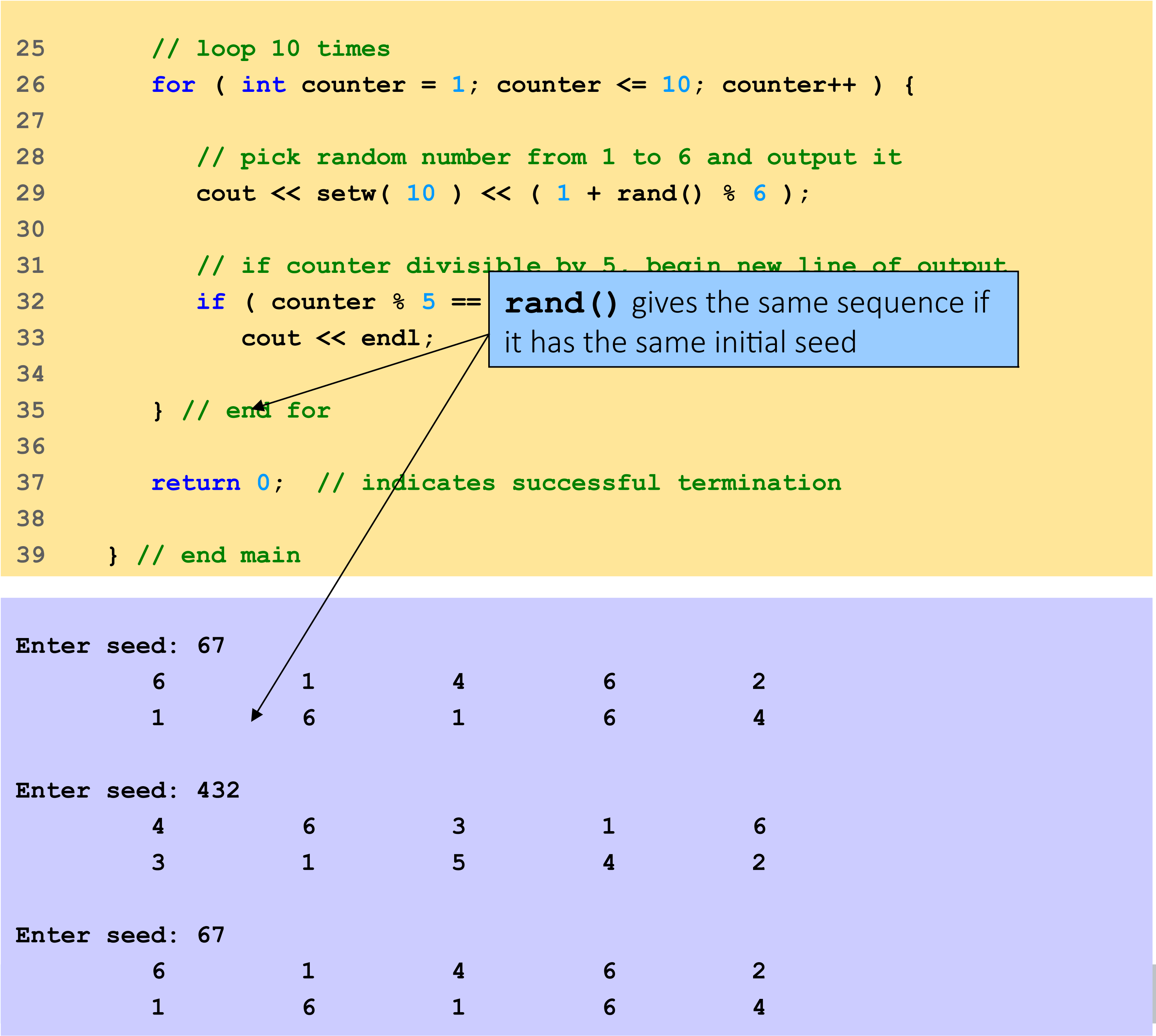
seed.cpp

Seed of the srand

```cpp
   // loop 10 times
   for ( int counter = 1; counter <= 10; counter++ ) {

      // pick random number from 1 to 6 and output it
      cout << setw( 10 ) << ( 1 + rand() % 6 );

      // if counter divisible by 5, begin new line of output
      if ( counter % 5 == 0 )
         cout << endl;

   } // end for

   return 0;  // indicates successful termination

} // end main
```

seed.cpp

Seed of the srand



```cpp
   // loop 10 times
   for ( int counter = 1; counter <= 10; counter++ ) {

      // pick random number from 1 to 6 and output it
      cout << setw( 10 ) << ( 1 + rand() % 6 );

      // if counter divisible by 5, begin new line of output
      if ( counter % 5 ==
         cout << endl;

   } // end for

   return 0;  // indicates successful termination

} // end main
```

rand() gives the same sequence if it has the same initial seed

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

Enter seed: 432
         4         6         3         1         6
         3         1         5         4         2

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

seed.cpp

Random Number Generation

- Can use the current time to set the seed
  - No need to explicitly set seed every time
  - **`srand( time( 0 ) );`**
  - **`time( 0 );`**
  - **`<ctime>`** **:** Returns current time in seconds

Recursion

- Recursive functions
  - Functions that call themselves
  - Can only solve a base case
- If not base case
  - Break problem into smaller problem(s)
  - Launch new copy of function to work on the smaller problem (recursive call/recursive step)
    - Slowly converges towards base case
    - Function makes call to itself inside the return statement
  - Eventually base case gets solved
    - Answer works way back up, solves entire problem

Recursion

- Example: factorial

  *n! = n * (n – 1) * (n – 2) * … * 1*

  - Recursive relationship (n! = n * (n – 1)!)

    *5! = 5 * 4!*

    *4! = 4 * 3!…*

  - Base case (1! = 0! = 1)

Example Using Recursion: Fibonacci Series

- Fibonacci series: 0, 1, 1, 2, 3, 5, 8...
  - Each number sum of two previous ones
  - Example of a recursive formula:

    fib(n) = fib(n-1) + fib(n-2)
  - C++ code for Fibonacci 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: Fibonacci Series

- Order of operations
  - `return fibonacci( n - 1 ) + fibonacci( n - 2 );`
- Do not know which one executed first
  - C++ does not specify
  - Only **`&&`**, **`||`** and **`?:`** guaranteed left-to-right evaluation
- Recursive function calls
  - Each level of recursion doubles the number of function calls
  - 30th number = 2^30 ~ 4 billion function calls
  - Exponential complexity

Esercitazione 4

- Simulate 6000 rolls of a dice and print the number of 1’s, 2’s, 3’s, etc. rolled. This should be roughly 1000 of each. (Dado.cpp)

```
./Dado
Face   Frequency
  1       980
  2       993
  3      1030
  4      1009
  5      1002
  6       986
```

- Write a program which evaluate the Fibonacci series using a recursive function. (fibonacci.cpp)

```
./fibonacci

    Enter an integer: 0                 Enter an integer: 2
    Fibonacci(0) = 0                    Fibonacci(2) = 1

    Enter an integer: 1                 Enter an integer: 3
    Fibonacci(1) = 1                    Fibonacci(3) = 2
```

Esercitazione 4

- Write a program that calculates the circumference and area of a circle. The input from keyboard is the radius. Use two functions, one to calculate the area and the other one to calculate the circumference of the circle.
- The two functions have to be implemented in an external macro (cerchio.{h,cpp}) and called in an external program (useCerchio.cpp)

Arrays

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
- 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]`**

Declaring Arrays

- When declaring arrays you have to specify
  - Name
    - Type of array : can be 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];
```

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 are set to 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

Using Arrays

```cpp
// Fig. 4.3: fig04_03.cpp
// Initializing an array.
#include <iostream>

using std::cout;
using std::endl;

#include <iomanip>

using std::setw;

int main()
{
   int n[ 10 ];  // n is an array of 10 integers

   // initialize elements of array n to 0
   for ( int i = 0; i < 10; i++ )
      n[ i ] = 0;   // set element at location i to 0

   cout << "Element" << setw( 13 ) << "Value" << endl;

   // output contents of array n in tabular format
   for ( int j = 0; j < 10; j++ )
      cout << setw( 7 ) << j << setw( 13 ) << n[ j ] << endl;

   return 0;  // indicates successful termination

} // end main
```

usearray1.cpp

Using Arrays

```cpp
// Fig. 4.3: fig04_03.cpp
// Initializing an array.
#include <iostream>

using std::cout;
using std::endl;

#include <iomanip>

using std::setw;

int main()
{
   int n[ 10 ];  // n is an array of

   // initialize elements of array n
   for ( int i = 0; i < 10; i++ )
      n[ i ] = 0;   // set element at location i to 0

   cout << "Element" << setw( 13 ) << "Value" << endl;

   // output contents of array n in tabular format
   for ( int j = 0; j < 10; j++ )
      cout << setw( 7 ) << j << setw( 13 ) << n[ j ] << endl;

   return 0;  // indicates successful termination

} // end main
```

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]**.

usearray1.cpp

Using Arrays

```cpp
// Fig. 4.3: fig04_03.cpp
// Initializing an array.
#include <iostream>

using std::cout;
using std::endl;

#include <iomanip>

using std::setw;

int main()
{
   int n[ 10 ];  // n is an array of 10 integers

   // initialize elements of array n to 0
   for ( int i = 0; i < 10; i++ )
      n[ i ] = 0;   // set element at location i to 0

   cout << "Element" << setw( 13 ) << "Value" << endl;

   // output contents of array n in tabular format
   for ( int j = 0; j < 10; j++ )
      cout << setw( 7 ) << j << setw( 13 ) << n[ j ] << endl;

   return 0;  // indicates successful termination

} // end main
```

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

usearray1.cpp

Using Arrays- 2

```cpp
 // 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;

int main()
{
   // use initializer list to initialize array n
   int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };

   cout << "Element" << setw( 13 ) << "Value" << endl;

   // output contents of array n in tabular format
   for ( int i = 0; i < 10; i++ )
      cout << setw( 7 ) << i << setw( 13 ) << n[ i ] << endl;

   return 0;  // indicates successful termination

} // end main
```

Note the use of the initializer list

usearray2.cpp

Using Arrays - 2

```cpp
 // 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;

int main()
{
   // use initializer list to initialize array n
   int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };

   cout << "Element" << setw( 13 ) << "Value" << endl;

   // output contents of array n in tabular format
   for ( int i = 0; i < 10; i++ )
      cout << setw( 7 ) << i << setw( 13 ) << n[ i ] << endl;

   return 0;  // indicates successful termination

} // end main
```

Note the use of the initializer list

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

usearray2.cpp

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

Using Arrays- 3

```cpp
 // Fig. 4.5: fig04_05.cpp
 // Initialize array s to the even integers from 2 to 20.
 #include <iostream>

 using std::cout;
 using std::endl;

 #include <iomanip>

using std::setw;

int main()
{
   // constant variable can be used to specify array size
   const int arraySize = 10;

   int s[ arraySize ];  // array s has 10 elements

   for ( int i = 0; i < arraySize; i++ )  // set the values
      s[ i ] = 2 + 2 * i;

   cout << "Element" << setw( 13 ) << "Value" << endl;

   // 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
```

usearray3.cpp

Using Arrays - 3

```cpp
 // Fig. 4.5: fig04_05.cpp
 // Initialize array s to the even integers from 2 to 20.
 #include <iostream>

 using std::cout;
 using std::endl;

 #include <iomanip>

using std::setw;

int main()
{
   // constant variable can be used to speci
   const int arraySize = 10;

   int s[ arraySize ];  // array s has 10 elements

   for ( int i = 0; i < arraySize; i++ )  // set the values
      s[ i ] = 2 + 2 * i;

   cout << "Element" << setw( 13 ) << "Value" << endl;

   // output contents of array s in tabular format
   for ( int j = 0; j < arraySize; j++ )
      cout << setw( 7 ) << j << setw( 13 ) << s[ j ] << e

   return 0;  // indicates successful termination

} // end main
```

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).

usearray3.cpp

Using Arrays- 3

```cpp
 // Fig. 4.5: fig04_05.cpp
 // Initialize array s to the even integers from 2 to 20.
 #include <iostream>

 using std::cout;
 using std::endl;

 #include <iomanip>

using std::setw;

int main()
{
   // constant variable can be used to specify array size
   const int arraySize = 10;

   int s[ arraySize ];  // array s has 10 elements

   for ( int i = 0; i < arraySize; i++ )  // set the values
      s[ i ] = 2 + 2 * i;

   cout << "Element" << setw( 13 ) << "Value" << endl;

   // 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
```

usearray3.cpp

Good use of const

```cpp
 // 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
```

goodconst.cpp

Bad use of const

```
1        // Fig. 4.7: fig04_07.cpp
2        // A const object must be initialized
3
4        int main()
5        {
6           const int x;  // Error: x must be initialized
7
8           x = 7;        // Error: cannot modify a const variable
9
10         return 0;     // indicates successful termination
11
12      } // end main
```

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

```
badconst.cpp: In function 'int main()':
badconst.cpp:6:15: error: uninitialized const 'x' [-fpermissive]
     const int x;  // Error: x must be initialized
               ^
badconst.cpp:8:7: error: assignment of read-only variable 'x'
     x = 7;        // Error: cannot modify a const variable
```

badconst.cpp

Using Arrays- 4

```
 // Fig. 4.8: fig04_08.cpp
 // Compute the sum of the elements of the array.
 #include <iostream>

 using std::cout;
 using std::endl;

 int main()
 {
   const int arraySize = 10;

   int a[ arraySize ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

   int total = 0;

   // sum contents of array a
   for ( int i = 0; i < arraySize; i++ )
      total += a[ i ];

   cout << "Total of array element values is " << total << endl;

   return 0;  // indicates successful termination

} // end main
```

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

usearray4.cpp

Examples Using Arrays

- Input as a “Strings” (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'`**

Examples Using Arrays : Input from keyboard

- 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 (BUT this is something we want to avoid)

- Printing strings

```
cout << string2 << endl;
```

    - Does not work for other array types
  - Characters printed until **`null`** found

String and input from keyboard

```cpp
 // 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 characters

   // 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";

   // 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
```

stringkeyboard.cpp

String and input from keyboard

```cpp
// 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 characters

   // 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";

   // 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
```

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

stringkeyboard.cpp

String and input from keyboard

```
 // 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 characters

   // read string from user into a
   cout << "Enter the string \"hel
   cin >> string1; // reads "hello" [space terminates input]

   // output strings
   cout << "string1 is: " << string1
        << "\nstring2 is: " << string2;

   cout << "\nstring1 with spaces between characters is:\n";

   // 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
```

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

stringkeyboard.cpp

String and input from keyboard

```
// 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 characters

   // 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";

   // 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
```

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

stringkeyboard.cpp

String and input from keyboard

```
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
```

stringkeyboard.cpp

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
- 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] );`**

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 [] );`**

Passing Arrays to Functions

```
// 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;

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

int main()
{
   const int arraySize = 5;                    // size of array a
   int a[ arraySize ] = { 0, 1, 2, 3, 4 };  // initialize a

   cout << "Effects of passing entire array by reference:"
        << "\n\nThe values of the original array are:\n";

   // output original array
   for ( int i = 0; i < arraySize; i++ )
      cout << setw( 3 ) << a[ i ];
```

Syntax for accepting an array in parameter list

arraytofunc.cpp

Passing Arrays to Functions

```

   cout << endl;

   // pass array a to modifyArray
   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
        << "\n\nThe value of a[3] is

   // pass array element a[ 3 ] by v
   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.

arraytofunc.cpp

Passing Arrays to Functions

```

// 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 loca
// 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.

arraytofunc.cpp

Passing Arrays to Functions

```
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
```

arraytofunc.cpp

Passing Arrays to Functions-2

```
// Fig. 4.15: fig04_15.cpp
// Demonstrating the const type qualifier.
#include <iostream>

using std::cout;
using std::endl;

void tryToModifyArray( const int [] );  // func

int main()
{
   int a[] = { 10, 20, 30 };

   tryToModifyArray( a );

   cout << a[ 0 ] << ' ' << a[ 1 ] << ' ' << a[ 2 ] << '\n';

   return 0;  // indicates successful termination

} // end main

// 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
```

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

arraytofunc2.cpp

Passing Arrays to Functions-2

```
arraytofunc2.cpp: In function ‘void tryToModifyArray(const int*)’:
arraytofunc2.cpp:26:10: error: assignment of read-only location ‘* b’
   b[ 0 ] /= 2;    // error
          ^
arraytofunc2.cpp:27:10: error: assignment of read-only location ‘*(b +
4u)’
   b[ 1 ] /= 2;    // error
          ^
arraytofunc2.cpp:28:10: error: assignment of read-only location ‘*(b +
8u)’
   b[ 2 ] /= 2;    // error
```

arraytofunc2.cpp

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

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)

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
```

Sorting Arrays

```
 // 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 ];

```

sortingarrays.cpp

Sorting Arrays

```
// 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
      // first element is greater than second e
      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.

sortingarrays.cpp

Sorting Arrays

```cpp
   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
```

sortingarrays.cpp

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)

Mean, Median and Mode Using Arrays

```
 // 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;

#include <iomanip>

using std::setw;
using std::setprecision;

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

int main()
{
   const int responseSize = 99;  // size of array responses

```

MeanMedianMode.cpp

Mean, Median and Mode Using Arrays

```cpp
   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,
            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 };

   // process responses
   mean( response, responseSize );
   median( response, responseSize );
   mode( frequency, response, responseSize );

   return 0;  // indicates successful termination

} // end main

```

MeanMedianMode.cpp

Mean, Median and Mode Using Arrays

```cpp
// 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 ];

   // format and output results
   cout << fixed << setprecision( 4 );

   cout << "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 (" << arraySize
        << "). The mean value for\nthis run is: "
        << total << " / " << arraySize << " = "
        << static_cast< double >( total ) / arraySize
        << "\n\n";

} // end function mean

```

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

MeanMedianMode.cpp

Mean, Median and Mode Using Arrays

```cpp
// sort array and determine median element's value
void median( int answer[], int size )
{
   cout << "\n********\n Median\n********\n"
        << "The unsorted array of responses is";

   printArray( answer, size );  // output unsorted array

   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

```

MeanMedianMode.cpp

Mean, Median and Mode Using Arrays

```
// 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";
```

MeanMedianMode.cpp

Mean, Median and Mode Using Arrays

```
   // output results
   for ( int rating = 1; rating <= 9; rating++ ) {
      cout << setw( 8 ) << rating << setw( 11 )
           << freq[ rating ] << "          ";

      // keep track of mode value and largest feque
      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;

} // end function mode
```

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

MeanMedianMode.cpp

Mean, Median and Mode Using Arrays

```cpp

// 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

```

MeanMedianMode.cpp

Mean, Median and Mode Using Arrays

```cpp
// 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
```

MeanMedianMode.cpp

Mean, Median and Mode Using Arrays

```
********
  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

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

MeanMedianMode.cpp

Mean, Median and Mode Using Arrays

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

MeanMedianMode.cpp

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

- 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^N > # of elements
    - 30 element array takes at most 5 steps

      $2^5 > 30$

Linear Search

```
 // 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

int main()
{
   const int arraySize = 100;  // size of array a
   int a[ arraySize ];         // create array a
   int searchKey;              // value to locate in a

   for ( int i = 0; i < arraySize; i++ )  // create some data
      a[ i ] = 2 * i;

   cout << "Enter integer search key: ";
   cin >> searchKey;

   // attempt to locate searchKey in array a
   int element = linearSearch( a, searchKey, arraySize );

```

Takes array, search key, and array size.

linearSearch.cpp

Linear Search

```
   // 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
```

```
Enter integer search key: 36
Found value in element 18

Enter integer search key: 37
Value not found
```

Binary Search

```
 // 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
      a[ i ] = 2 * i;

```

binarySearch.cpp

Binary Search

```
   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

```

binarySearch.cpp

Binary Search

```cpp
// 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 greate
   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

binarySearch.cpp

Binary Search

```
         // if searchKey matches middle element, return middle
         if ( searchKey == b[ middle ] )  // match
            return middle;

         else

            // if searchKey less than middle elem
            // set new high element
            if ( searchKey < b[ middle ] )
               high = middle - 1;  // search low end of array

            // if searchKey greater than middl
            // set new low element
            else
               low = middle + 1;   // search h
      }

      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.

binarySearch.cpp

Binary Search

```

// 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

```

binarySearch.cpp

Binary Search

```
// 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 )
         cout << "    ";

      // display middle element marked with a *
      else

         if ( m == mid )           // mark middle value
            cout << setw( 3 ) << b[ m ] << '*';

         // display other elements in subarray
         else
            cout << setw( 3 ) << b[ m ] << ' ';

   cout << endl;  // start new line of output

} // end function printRow
```

binarySearch.cpp

Binary Search

```
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: 25

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
                                 16  18  20  22* 24  26  28
                                                 24  26* 28
                                                 24*

25 not found
```

binarySearch.cpp

Binary Search

```
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
```

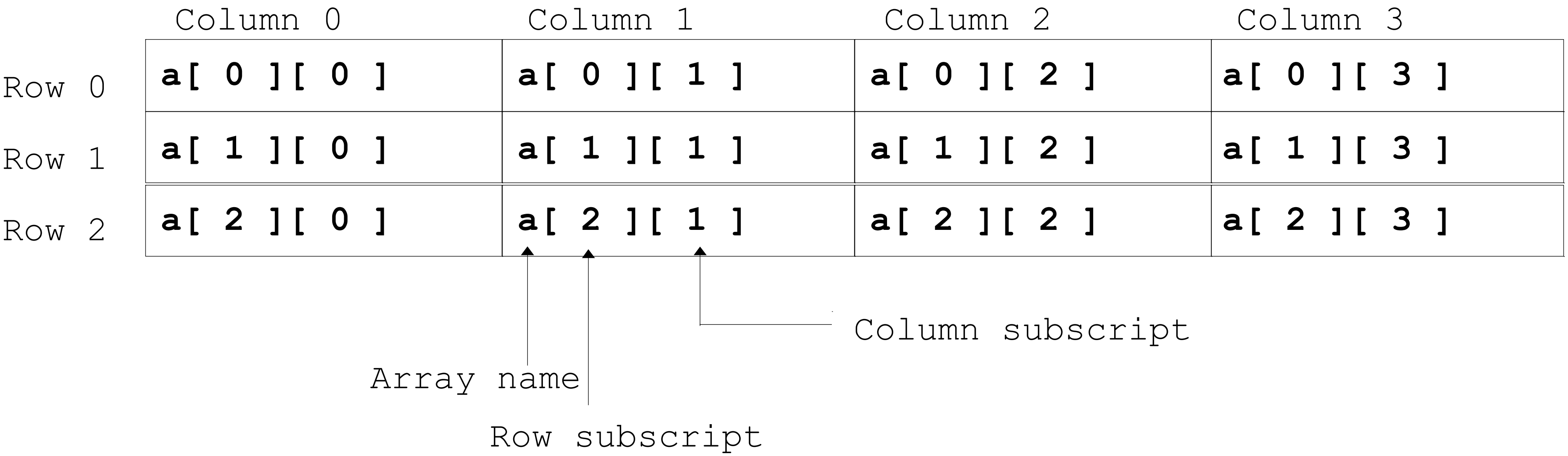
binarySearch.cpp

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

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

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 ] );
  ```

Multiple-Subscripted Arrays

```cpp
// 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 );

   return 0;  // indicates successful termination

} // end main
```

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.

matrix.cpp

Multiple-Subscripted Arrays

```

// function to output array with two rows
void printArray( int a[][ 3 ] )
{
   for ( int i = 0; i < 2; i++ ) {    // f

      for ( int j = 0; j < 3; j++ )   // output column values
         cout << a[ i ][ j ] << ' ';

      cout << endl;  // start new line of output

   } // end outer for structure

} // end function printArray
```

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
```

matrix.cpp

Multiple-Subscripted Arrays

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

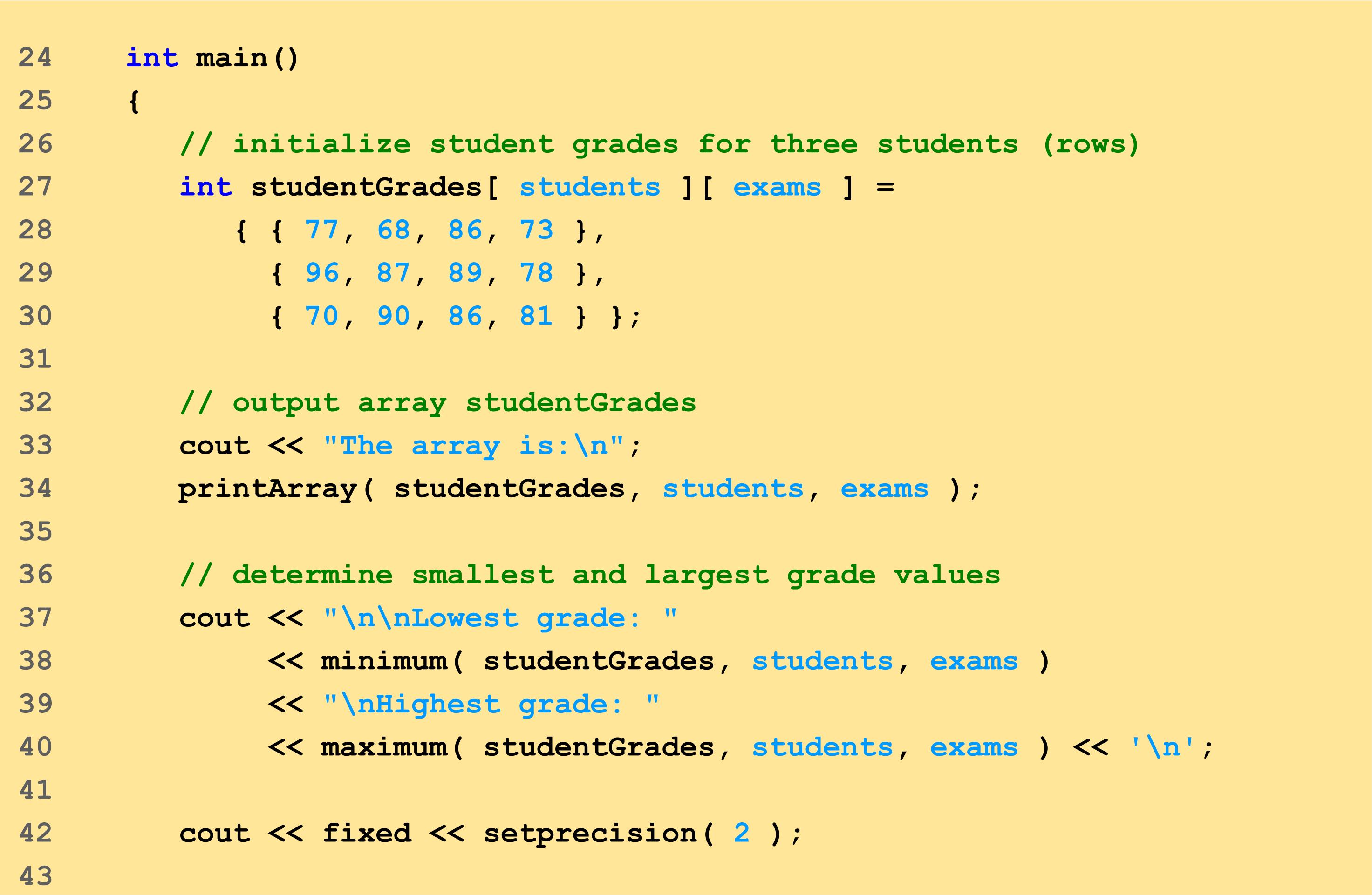
	Quiz1	Quiz2
Student0	95	85
Student1	89	80

Multiple-Subscripted Arrays

```cpp
 // 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 );

```

QuizArray.cpp

Multiple-Subscripted Arrays



```cpp
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 );

```

QuizArray.cpp

Multiple-Subscripted Arrays

```
   // 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 termination

} // 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;

} // end function minimum
```

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

QuizArray.cpp

Multiple-Subscripted Arrays

```cpp

// 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

// 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
```

QuizArray.cpp

Multiple-Subscripted Arrays

```
// 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
```

QuizArray.cpp

Multiple-Subscripted Arrays

```
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
```

QuizArray.cpp

Esercitazione 5

1) Copy the files sortArray.{h,cpp} into your working directory.

The files contain statements and definitions of some useful functions to manipulate array.

Write a program that generates N (chosen by the user) random numbers between 1 and Nmax (chosen by the user) and stores them in a matrix of size N. Using the functions contained in the array handle:

- print the generated matrix
- perform average, median and mode calculations of the elements of the matrix
- print the histogram of the frequencies of the individual values

Esercitazione 5

Execution example:

```
./usesortArray

 Tell me the maximum random number you want to generate 15
 Tell me the size of the Array of random numbers 100
 Here is the matrix:
 16  1  11  11  10  15   9   9  16  11   7  10  14  15   7   3  16   9  14   5    1   2   8   2   4   6   3   5   8  13  15   7  13   9   1   6   7   9  15   7    3   5  16   1   3   7   3   2  15  16   6  15
1  13  16   4   3   2   9  10   15   7  16  11  15  16   1   6   8  15  12  11   3  11  11   5   1  13   6  15   13  12  13  13   8  12   1  10  14   9   3  12  15   2   6  13   1   6   2   9

 Average is: 8.72
 Median is : 9
 Mode is   : 15

 Histogram of the frequencies:
 0
 1 *********
 2 ******
 3 ********
 4 **
 5 ****
 6 *******
 7 *******
 8 ****
 9 ********
10 ****
11 *******
12 ****
13 ********
14 ***
15 ***********
```

Esercitazione 5

2) Copy the files searchArray.{h,cpp}. They contain functions for the linear and binary search of a given value in an array. Write a program that generates an array and looks for a value chosen by the user in the created array with the linear search method.

```
g++ searchArray.cpp useLinearSearchArray.cpp-o useLinearSearchArray
./useLinearSearchArray
```

3) Write a program that generates an array and looks for a value chosen by the user in that array with the binary search method. Remember that the matrix must be ordered. Use the bubbleSort function implemented in sortArray.cpp function for this purpose.

```
g++ searchArray.cpp sortArray.cpp useBinarySearchArray.cpp-o
useBinarySearchArray
./useBinarySearchArray
```