Functions

Functions

- To make large programs manageable, programmers modularize them into subprograms called functions
- They can be compiled and tested separately and reused in different programs.
- This “modularization” is characteristic of “object-oriented” software
- The Standard C++ Library is a collection of pre-defined functions and other program elements which are accessed through header files (like `<iostream>` that we used on Monday)
- Let’s focus now on other Standard C++ Library (`<cmath>`, `<cstdlib>`,`<ctime>`)

Math Library Functions

- Perform common mathematical calculations
  - To access math library functions 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
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*	sin(0.0) is 0

http://www.cplusplus.com/reference/cmath/

Absolute value (abs)

In C++, `std::abs` is overloaded for both signed integer and floating point types.

`std::fabs` only deals with floating point types (pre C++11).

Note that the std:: is important, the C function ::abs that is commonly available for legacy reasons will only handle int!

The problem with

```
float f2= fabs(-9);
```

is not that there is no conversion from int (the type of-9) to double, but that the compiler does not know which conversion to pick (`int -> float, double, long double`) since there is a `std::fabs` for each of those three. Your workaround explicitly tells the compiler to use the int-> double conversion, so the ambiguity goes away.

C++11 solves this by adding double `fabs( Integral arg );` which will return the abs of any integer type converted to double. Apparently, this overload is also available in C++98 mode with libstdc++ and libc++.

In general, just use `std::abs`, it will do the right thing.

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

```
1   // Fig. 3.3: fig03_03.cpp
2   // Creating and using a programmer-defined function.
3   #include <iostream>
4
5   using std::cout;
6   using std::endl;
7
8   int square( int );   // function prototype
9
10  int main()
11  {
12    // loop 10 times and calculate and output
13    // square of x each time
14    for ( int x = 1; x <= 10; x++ )
15       cout << square( x ) << "  ";  // function call
16
17    cout << endl;
18
19    return 0;  // indicates successful termination
20
21  } // end main
22
23 // square function definition returns square of an integer
24 int square( int y )  // y is a copy of argument to function
25 {
26    return y * y;     // returns square of y as an int
27
28 } // end function square
```

squareFunc.cpp

Square function

```
1   // Fig. 3.3: fig03_03.cpp
2   // Creating and using a programmer-defined function.
3   #include <iostream>
4
5   using std::cout;
6   using std::endl;
7
8   int square( int );   // function prototype
9
10  int main()
11  {
12    // loop 10 times and calculate and
13    // square of x each time
14    for ( int x = 1; x <= 10; x++ )
15       cout << square( x ) << "  ";  // function call
16
17    cout << endl;
18
19    return 0;  // indicates successful termination
20
21  } // end main
22
23 // square function definition returns square of an integer
24 int square( int y )  // y is a copy of argument to function
25 {
26    return y * y;     // returns square of y as an int
27
28 } // end function square
```

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

squareFunc.cpp

Square function

```
1   // Fig. 3.3: fig03_03.cpp
2   // Creating and using a programmer-defined function.
3   #include <iostream>
4
5   using std::cout;
6   using std::endl;
7
8   int square( int );   // function prototype
9
10  int main()
11  {
12    // loop 10 times and calculate and output
13    // square of x each time
14    for ( int x = 1; x <= 10; x++ )
15       cout << square( x ) << "  ";  // function call
16
17    cout << endl;
18
19    return 0;  // indicates successful termination
20
21  } // end main
22
23  // square function definition returns square of an integer
24  int square( int y ) // y is a copy of argument to function
25  {
26     return y * y;     // returns square o
27
28  } // end function square
```

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

squareFunc.cpp

Square function

```
1   // Fig. 3.3: fig03_03.cpp
2   // Creating and using a programmer-defined function.
3   #include <iostream>
4
5   using std::cout;
6   using std::endl;
7
8   int square( int );   // function prototype
9
10  int main()
11  {
12    // loop 10 times and calculate and output
13    // square of x each time
14    for ( int x = 1; x <= 10; x++ )
15       cout << square( x ) << "  ";  // function call
16
17    cout << endl;
18
19    return 0;  // indicates successful termination
20
21  } // end main
22
23 // square function definition returns square of an integer
24 int square( int y )  // y is a copy of argument to function
25 {
26    return y * y;     // returns square of y as an int
27
28 } // end function square
```

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

squareFunc.cpp

Maximum function

```
1   // Fig. 3.4: fig03_04.cpp
2   // Finding the maximum of three floating-point numbers.
3   #include <iostream>
4
5   using std::cout;
6   using std::cin;
7   using std::endl;
8
9   double maximum( double, double, double ); // function prototype
10
11  int main()
12  {
13     double number1;
14     double number2;
15     double number3;
16
17     cout << "Enter three floating-point numbers: ";
18     cin >> number1 >> number2 >> number3;
19
20     // number1, number2 and number3 are arguments to
21     // the maximum function call
22     cout << "Maximum is: "
23          << maximum( number1, number2, number3 ) << endl;
24
25     return 0;  // indicates successful termination
```

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

maximumFunc.cpp

Maximum function

```
26
27    } // end main
28
29    // function maximum definition;
30    // x, y and z are parameters
31    double maximum( double x, double y, double z )
32    {
33       double max = x;    // assume x is largest
34
35       if ( y > max )     // if y is larger,
36          max = y;        // assign y to max
37
38       if ( z > max )     // if z is larger,
39          max = z;        // assign z to max
40
41       return max;        // max is largest value
42
43    } // 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

Passing by reference

- Until now, all the parameters have been ***passed by value*** → the expression used in the function-call is at first evaluated and then the resulting value is assigned to the corresponding parameter in the function's parameter list before the function begins executing.
- The pass-by-value mechanism allows for a more general expressions to be used in place of an argument in the function call. For example the `cube()` function could also be called as `cube(3)`, or as `cube(2*x-3)`, or even as `cube(2*sqrt(x)-cube(3))`.
  - In each case, the expression within the parentheses is evaluated to a single value and then that value is passed to the function.
- The *read-only, pass-by-value* method of communication is usually what we want for functions. It makes the functions more self-contained, protecting them against accidental side effects.
- There are some situations where a function needs to change the value of the parameter passed to it. That can be done by passing it ***by reference***.
- To pass a parameter by reference instead of by value, simply append an ampersand, **&**, to the type specifier in the functions parameter list.
- This makes the local variable *a reference* to the argument passed to it. So the argument is *read-write* instead of *read-only*. Any change to the local variable inside the function will cause the same change to the argument that was passed to it.

Passing by reference

```
#include <iostream>

using namespace std;

void passbyvalue(int x);
void passbyreference(int& x);

int main(){

  int var1 = 5;
  int var2 = 5;

  cout<<"Inital var1 : "<<var1<<endl;
  cout<<"Inital var2 : "<<var2<<endl;

  passbyvalue(var1);
  passbyreference(var2);

  cout<<"Function by value     var1 : "<<var1<<endl;
  cout<<"Function by reference var2 : "<<var2<<endl;

}

void passbyvalue(int x){
  x = 7;
}

void passbyreference(int &x){
  x = 7;
}
```

PassByReference.cpp

Passing by reference

```
#include <iostream>

using namespace std;

void passbyvalue(int x);
void passbyreference(int& x);

int main(){

  int var1 = 5;
  int var2 = 5;

  cout<<"Inital var1 : "<<var1<<endl;
  cout<<"Inital var2 : "<<var2<<endl;

  passbyvalue(var1);
  passbyreference(var2);

  cout<<"Function by value     var1 : "<<var1<<endl;
  cout<<"Function by reference var2 : "<<var2<<endl;

}

void passbyvalue(int x){
  x = 7;
}

void passbyreference(int &x){
  x = 7;
}
```

```
using namespace std;
```

Is it bad?

C++ has a standard library that contains common functionality you use in building your applications like containers, algorithms, etc. If names used by these were out in the open, for example, if they defined a queue class globally, you'd never be able to use the same name again without conflicts. So they created a namespace, std to contain this change.

The using namespace statement just means that in the scope it is present, make all the things under the std namespace available without having to prefix std:: before each of them.

While this practice is okay for example code, pulling in the entire std namespace into the global namespace is not good as it defeats the purpose of namespaces and can lead to name collisions. This situation is called namespace pollution.

→ Do not do like me! And instead use

```
using std::cout;

using std::endl;

...
```

PassByReference.cpp

Passing by reference

```
#include <iostream>

using namespace std;

void passbyvalue(int x);
void passbyreference(int& x);

int main(){

  int var1 = 5;
  int var2 = 5;

  cout<<"Inital var1 : "<<var1<<endl;
  cout<<"Inital var2 : "<<var2<<endl;

  passbyvalue(var1);
  passbyreference(var2);

  cout<<"Function by value     var1 : "<<var1<<endl;
  cout<<"Function by reference var2 : "<<var2<<endl;

}

void passbyvalue(int x){
  x = 7;
}

void passbyreference(int &x){
  x = 7;
}
```

```
Initial var1 : 5
Initial var2 : 5
Function by value     var1 : 5
Function by reference var2 : 7
```

PassByReference.cpp

Header Files

- Header files contain
  - Function prototypes
  - Definitions of data types and constants
- Library header files

```
#include <cmath>
```

- Header files ending with .h
  - Programmer-defined header files

```
#include "myheader.h"
```

Function definition in an external macro

- Example: Create a library of functions 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
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

Overloading function

- C++ allows you to use the same name for different functions.
- As long as they have different parameter type lists, the compiler will regard them as different functions.
- To be distinguished, the parameter lists must either contain a different number of parameters, or there must be at least one position in their parameter lists where the types are different.

Overloading `max()` function

```
#include <iostream>

using std::cout;  // program uses cout
using std::endl;  // program uses endl

int max(int, int);
int max(int, int, int);

int main(){
  cout<<"Max(10,20): " <<max(10,20)<<"\nMax(2,9,6): "<<max(2,9,6)<<endl;
  return 0;
}

int max(int x, int y){
  return (x>y ? x: y);
}

int max(int x, int y, int z){
  int m = (x>y ? x: y); // m = max(x,y)
  return (z>m ? z: m);
}
```

```
Max(10,20): 20
Max(2,9,6): 9
```

OverloadingMax.cpp

Command line arguments in C/C++

- The most important function of C/C++ is **`main()`** function. It is mostly defined with a return type of **`int`** and without parameters :

```
int main() { /* ... */ return 0; }
```

- We can also give command-line arguments in C and C++. Command-line arguments are given after the name of the program in command-line shell of Operating Systems.
- To pass command line arguments, we typically define `main()` with two arguments: first argument is the number of command line arguments and second is list of command-line arguments.

```
int main(int argc, char *argv[]) { /* ... */ }
```

or

```
int main(int argc, char **argv) { /* ... */ }
```

- **`argc`** (ARGument Count) is a **`int`** and stores number of command-line arguments passed by the user including the name of the program. So if we pass a value to a program, value of argc would be 2 (one for argument and one for program name)
- The value of argc should be non negative.
- **`argv`**(ARGument Vector) is array of character pointers listing all the arguments.
  - If argc is greater than zero,the array elements from argv[0] to argv[argc-1] will contain pointers to strings.
  - `argv[0]` is the name of the program , After that till argv[argc-1] every element is command-line arguments.

Command line arguments in C/C++

```
#include <iostream>
#include <cstdlib>

using std::cout;  // program uses cout
using std::endl;  // program uses endl

int max(int, int);
int max(int, int, int);

int main(int argc, char* argv[]){

  int a = atoi(argv[1]);
  int b = atoi(argv[2]);
  int c = atoi(argv[3]);
  // Converting string type to integer type
  // using function "atoi( argument)"

  cout<<"Max("<<a<<","<<b<<"): " <<max(a,b)<<endl;
  cout<<"Max("<<a<<","<<b<<","<<c<<"): "<<max(a,b,c)<<endl;
  return 0;
}

int max(int x, int y){
  return (x>y ? x: y);
}

int max(int x, int y, int z){
  int m = (x>y ? x: y); // m = max(x,y)
  return (z>m ? z: m);
}
```

OverloadingMaxArg.cpp

Command line arguments in C/C++

```
g++ OverloadingMaxArg.cpp -o max
./max 4 5 6
Max(4,5): 5
Max(4,5,6): 6
```

Important: if you define a function that needs some argument, you have to specify them at the runtime, otherwise you will get a segmentation violation!!!

Some addition standard library function

Random Number Generation

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

- stdlib.h functions can be classified into 5 categories
- Conversion between types : Atof, atoi, atol, strtod, strtoul
- Allocation and de-allocation of memory: calloc, malloc, realloc
- Processes control : free, abort
- Simple math functions: abs, labs, div, ldiv
- Pseudo-random number generation : rand , srand

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

```
1   // Fig. 3.7: fig03_07.cpp
2   // Shifted, scaled integers produced by 1 + rand() % 6.
3   #include <iostream>
4
5   using std::cout;
6   using std::endl;
7
8   #include <iomanip>
9
10  using std::setw;
11
12  #include <cstdlib>   // contains function prototype for rand
13
14  int main()
15  {
16     // loop 20 times
17     for ( int counter = 1; counter <= 20; counter++ ) {
18
19        // pick random number from 1 to 6 and output it
20        cout << setw( 10 ) << ( 1 + rand() % 6 );
21
22        // if counter divisible by 5, begin new line of output
23        if ( counter % 5 == 0 )
24           cout << endl;
25
26     } // end for structure
27
28     return 0;  // indicates successful termination
29
30  } // end main
```

Roll dice

rolldice.cpp

```
1   // Fig. 3.7: fig03_07.cpp
2   // Shifted, scaled integers produced by 1 + rand() % 6.
3   #include <iostream>
4
5   using std::cout;
6   using std::endl;
7
8   #include <iomanip>
9
10  using std::setw;
11
12  #include <cstdlib>   // contains function prototype for rand
13
14  int main()
15  {
16     // loop 20 times
17     for ( int counter = 1; counter <= 20; counter++ )
18
19        // pick random number from 1 to 6 and output it
20        cout << setw( 10 ) << ( 1 + rand() % 6 );
21
22        // if counter divisible by 5, begin new line of output
23        if ( counter % 5 == 0 )
24           cout << endl;
25
26     } // end for structure
27
28     return 0;  // indicates successful termination
29
30  } // end main
```

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

Roll dice

rolldice.cpp

```
1   // Fig. 3.7: fig03_07.cpp
2   // Shifted, scaled integers produced by 1 + rand() % 6.
3   #include <iostream>
4
5   using std::cout;
6   using std::endl;
7
8   #include <iomanip>
9
10  using std::setw;
11
12  #include <cstdlib>   // contains function prototype for rand
13
14  int main()
15  {
16     // loop 20 times
17     for ( int counter = 1; counter <= 20; counter++ ) {
18
19        // pick random number from 1 to 6 and output it
20        cout << setw( 10 ) << ( 1 + rand() % 6 );
21
22        // if counter divisible by 5, begin new line of output
23        if ( counter % 5 == 0 )
24           cout << endl;
25
26     } // end for structure
27
28     return 0;  // indicates successful termination
29
30  } // 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

```
1      // Fig. 3.9: fig03_09.cpp
2      // Randomizing die-rolling program.
3      #include <iostream>
4
5      using std::cout;
6      using std::cin;
7      using std::endl;
8
9      #include <iomanip>
10
11    using std::setw;
12
13    // contains prototypes for functions srand and rand
14    #include <cstdlib>
15
16    // main function begins program execution
17    int main()
18    {
19       unsigned seed;
20
21       cout << "Enter seed: ";
22       cin >> seed;
23       srand( seed );  // seed random number generator
24
```

seed.cpp

Seed of the srand

```
1      // Fig. 3.9: fig03_09.cpp
2      // Randomizing die-rolling program.
3      #include <iostream>
4
5      using std::cout;
6      using std::cin;
7      using std::endl;
8
9      #include <iomanip>
10
11     using std::setw;
12
13     // contains prototypes for functions srand and rand
14     #include <cstdlib>
15
16     // main function begins program execution
17     int main()
18     {
19        unsigned seed;
20
21        cout << "Enter seed: ";
22        cin >> seed;
23        srand( seed );  // seed random number generator
24
```

Setting the seed with **srand()**

seed.cpp

Seed of the srand

```
25       // loop 10 times
26       for ( int counter = 1; counter <= 10; counter++ ) {
27
28          // pick random number from 1 to 6 and output it
29          cout << setw( 10 ) << ( 1 + rand() % 6 );
30
31          // if counter divisible by 5, begin new line of output
32          if ( counter % 5 == 0 )
33             cout << endl;
34
35       } // end for
36
37       return 0;  // indicates successful termination
38
39    } // end main
```

seed.cpp

Seed of the srand

```
25       // loop 10 times
26       for ( int counter = 1; counter <= 10; counter++ ) {
27
28          // pick random number from 1 to 6 and output it
29          cout << setw( 10 ) << ( 1 + rand() % 6 );
30
31          // if counter divisible by 5, begin new line of output
32          if ( counter % 5 ==
33             cout << endl;
34
35       } // end for
36
37       return 0;  // indicates successful termination
38
39    } // 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 function

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

  $n! = n * (n - 1) * (n - 2) * \ldots * 1$

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

    $5! = 5 * 4!$

    $4! = 4 * 3!\ldots$

  - 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

Function- Exercises

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)
- Write a macro where the function min(a,b) and min(a,b,c) are overloaded. The argument of the function can be passed by keyboard with the cin operator.
- Repeat the exercise passing the argument from the command line.

```
./min
Insert how many numbers you want to compare
2
Insert the first number: 3
Insert the second number: 6
The minimum is: 3

./minArg 3 3 4 6
Min(3,4,6): 3
```

Esercitazione 4

- Write and test the computeTriangle() function that returns the area a and the perimeter p of a triangle with given size x,y and z:

```
void computeTriangle(float & a, float &p, float x, float y, float z)
```

To compute the area use Heron's formula

```
./main
Enter the sides:
2
3
5
The perimeter of the triangle is : 10 and its area is: 5
```

Esercitazione 4

- Simulate 6000 rolls of a dice and print the number of 1's, 2's, 3's, etc. rolled. There 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
Fibonacci(0) = 0

Enter an integer: 1
Fibonacci(1) = 1

Enter an integer: 2
Fibonacci(2) = 1

Enter an integer: 3
Fibonacci(3) = 2

Arrays

Arrays

- Array:
  - Sequence of object all of which have the same name and type (**`int`**, **`char`**, etc.).
  - The objects are called ***elements*** of the array and are numbered consecutively from **0**
- To refer to an element:
  - Specify array name and position number (***index***)
    - Format: arrayname[position number]
  - N-element array c
    - **`c[0]`**, **`c[1]`** … **`c[n - 1]`**
    - Nth element as position N-1
- Array elements are 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]`**
- An array can also be defined as a consecutive group of memory locations

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

```
1     // Fig. 4.3: fig04_03.cpp
2     // Initializing an array.
3     #include <iostream>
4
5     using std::cout;
6     using std::endl;
7
8     #include <iomanip>
9
10    using std::setw;
11
12    int main()
13    {
14       int n[ 10 ];  // n is an array of 10 integers
15
16       // initialize elements of array n to 0
17       for ( int i = 0; i < 10; i++ )
18          n[ i ] = 0;   // set element at location i to 0
19
20       cout << "Element" << setw( 13 ) << "Value" << endl;
21
22       // output contents of array n in tabular format
23       for ( int j = 0; j < 10; j++ )
24          cout << setw( 7 ) << j << setw( 13 ) << n[ j ] << endl;
25
26       return 0;  // indicates successful termination
27
28    } // end main
```

usearray1.cpp

Using Arrays

```
1   // Fig. 4.3: fig04_03.cpp
2   // Initializing an array.
3   #include <iostream>
4
5   using std::cout;
6   using std::endl;
7
8   #include <iomanip>
9
10  using std::setw;
11
12  int main()
13  {
14     int n[ 10 ];  // n is an array of 10 integers
15
16     // initialize elements of array n to 0
17     for ( int i = 0; i < 10; i++ )
18        n[ i ] = 0;   // set element at location i to 0
19
20     cout << "Element" << setw( 13 ) << "Value" << endl;
21
22     // output contents of array n in tabular format
23     for ( int j = 0; j < 10; j++ )
24        cout << setw( 7 ) << j << setw( 13 ) << n[ j ] << endl;
25
26     return 0;  // indicates successful termination
27
28  } // end main
```

setw : Sets the field width to be used on output operations
The function is included in the **<iomanip>** library

usearray1.cpp

Using Arrays

```
1   // Fig. 4.3: fig04_03.cpp
2   // Initializing an array.
3   #include <iostream>
4
5   using std::cout;
6   using std::endl;
7
8   #include <iomanip>
9
10  using std::setw;
11
12  int main()
13  {
14     int n[ 10 ];  // n is an array of
15
16     // initialize elements of array n
17     for ( int i = 0; i < 10; i++ )
18        n[ i ] = 0;   // set element at location i to 0
19
20     cout << "Element" << setw( 13 ) << "Value" << endl;
21
22     // output contents of array n in tabular format
23     for ( int j = 0; j < 10; j++ )
24        cout << setw( 7 ) << j << setw( 13 ) << n[ j ] << endl;
25
26     return 0;  // indicates successful termination
27
28  } // 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

```
1   // Fig. 4.3: fig04_03.cpp
2   // Initializing an array.
3   #include <iostream>
4
5   using std::cout;
6   using std::endl;
7
8   #include <iomanip>
9
10  using std::setw;
11
12  int main()
13  {
14     int n[ 10 ];  // n is an array of 10 integers
15
16     // initialize elements of array n to 0
17     for ( int i = 0; i < 10; i++ )
18        n[ i ] = 0;   // set element at location i to 0
19
20     cout << "Element" << setw( 13 ) << "Value" << endl;
21
22     // output contents of array n in tabular format
23     for ( int j = 0; j < 10; j++ )
24        cout << setw( 7 ) << j << setw( 13 ) << n[ j ] << endl;
25
26     return 0;  // indicates successful termination
27
28  } // 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

```
1   // Fig. 4.4: fig04_04.cpp
2   // Initializing an array with a declaration.
3   #include <iostream>
4
5   using std::cout;
6   using std::endl;
7
8   #include <iomanip>
9
10  using std::setw;
11
12  int main()
13  {
14     // use initializer list to initialize array n
15     int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
16
17     cout << "Element" << setw( 13 ) << "Value" << endl;
18
19     // output contents of array n in tabular format
20     for ( int i = 0; i < 10; i++ )
21        cout << setw( 7 ) << i << setw( 13 ) << n[ i ] << endl;
22
23     return 0;  // indicates successful termination
24
25  } // end main
```

Note the use of the initializer list

usearray2.cpp

Using Arrays- 2

```
1   // Fig. 4.4: fig04_04.cpp
2   // Initializing an array with a declaration.
3   #include <iostream>
4
5   using std::cout;
6   using std::endl;
7
8   #include <iomanip>
9
10  using std::setw;
11
12  int main()
13  {
14     // use initializer list to initialize array n
15     int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
16
17     cout << "Element" << setw( 13 ) << "Value" << endl;
18
19     // output contents of array n in tabular format
20     for ( int i = 0; i < 10; i++ )
21        cout << setw( 7 ) << i << setw( 13 ) << n[ i ] << endl;
22
23     return 0;  // indicates successful termination
24
25  } // 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

```
1     // Fig. 4.5: fig04_05.cpp
2     // Initialize array s to the even integers from 2 to 20.
3     #include <iostream>
4
5     using std::cout;
6     using std::endl;
7
8     #include <iomanip>
9
10    using std::setw;
11
12    int main()
13    {
14       // constant variable can be used to specify array size
15       const int arraySize = 10;
16
17       int s[ arraySize ];  // array s has 10 elements
18
19       for ( int i = 0; i < arraySize; i++ )  // set the values
20          s[ i ] = 2 + 2 * i;
21
22       cout << "Element" << setw( 13 ) << "Value" << endl;
23
24       // output contents of array s in tabular format
25       for ( int j = 0; j < arraySize; j++ )
26          cout << setw( 7 ) << j << setw( 13 ) << s[ j ] << endl;
27
28       return 0;  // indicates successful termination
29
30    } // end main
```

usearray3.cpp

Using Arrays- 3

```
1   // Fig. 4.5: fig04_05.cpp
2   // Initialize array s to the even integers from 2 to 20.
3   #include <iostream>
4
5   using std::cout;
6   using std::endl;
7
8   #include <iomanip>
9
10  using std::setw;
11
12  int main()
13  {
14     // constant variable can be used to speci
15     const int arraySize = 10;
16
17     int s[ arraySize ];  // array s has 10 elements
18
19     for ( int i = 0; i < arraySize; i++ )  // set the values
20        s[ i ] = 2 + 2 * i;
21
22     cout << "Element" << setw( 13 ) << "Value" << endl;
23
24     // output contents of array s in tabular format
25     for ( int j = 0; j < arraySize; j++ )
26        cout << setw( 7 ) << j << setw( 13 ) << s[ j ] << e
27
28     return 0;  // indicates successful termination
29
30  } // 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

```
1   // Fig. 4.5: fig04_05.cpp
2   // Initialize array s to the even integers from 2 to 20.
3   #include <iostream>
4
5   using std::cout;
6   using std::endl;
7
8   #include <iomanip>
9
10  using std::setw;
11
12  int main()
13  {
14     // constant variable can be used to specify array size
15     const int arraySize = 10;
16
17     int s[ arraySize ];  // array s has 10 elements
18
19     for ( int i = 0; i < arraySize; i++ )  // set the values
20        s[ i ] = 2 + 2 * i;
21
22     cout << "Element" << setw( 13 ) << "Value" << endl;
23
24     // output contents of array s in tabular format
25     for ( int j = 0; j < arraySize; j++ )
26        cout << setw( 7 ) << j << setw( 13 ) << s[ j ] << endl;
27
28     return 0;  // indicates successful termination
29
30  } // 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`

```
1       // Fig. 4.6: fig04_06.cpp
2       // Using a properly initialized constant variable.
3       #include <iostream>
4
5       using std::cout;
6       using std::endl;
7
8       int main()
9       {
10        const int x = 7;  // initialized constant variable
11
12        cout << "The value of constant variable x is: "
13             << x << endl;
14
15        return 0;  // indicates successful termination
16
17    } // 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

```
1       // Fig. 4.8: fig04_08.cpp
2       // Compute the sum of the elements of the array.
3       #include <iostream>
4
5       using std::cout;
6       using std::endl;
7
8       int main()
9       {
10        const int arraySize = 10;
11
12        int a[ arraySize ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
13
14        int total = 0;
15
16        // sum contents of array a
17        for ( int i = 0; i < arraySize; i++ )
18           total += a[ i ];
19
20        cout << "Total of array element values is " << total << endl;
21
22        return 0;  // indicates successful termination
23
24    } // end main
```

usearray4.cpp

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

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 as for “number array”

`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 white-space 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

```
1   // Fig. 4_12: fig04_12.cpp
2   // Treating character arrays as strings.
3   #include <iostream>
4
5   using std::cout;
6   using std::cin;
7   using std::endl;
8
9   int main()
10  {
11     char string1[ 20 ];                  // reserves 20 characters
12     char string2[] = "string literal"; // reserves 15 characters
13
14     // read string from user into array string2
15     cout << "Enter the string \"hello there\": ";
16     cin >> string1;  // reads "hello" [space terminates input]
17
18     // output strings
19     cout << "string1 is: " << string1
20          << "\nstring2 is: " << string2;
21
22     cout << "\nstring1 with spaces between characters is:\n";
23
24     // output characters until null character is reached
25     for ( int i = 0; string1[ i ] != '\0'; i++ )
26        cout << string1[ i ] << ' ';
27
28     cin >> string1;  // reads "there"
29     cout << "\nstring1 is: " << string1 << endl;
30
31     return 0;  // indicates successful termination
32
33  } // end main
```

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

stringkeyboard.cpp

String and input from keyboard

```
1   // Fig. 4_12: fig04_12.cpp
2   // Treating character arrays as strings.
3   #include <iostream>
4
5   using std::cout;
6   using std::cin;
7   using std::endl;
8
9   int main()
10  {
11     char string1[ 20 ];                  // reserves 20 characters
12     char string2[] = "string literal"; // reserves 15 characters
13
14     // read string from user into a
15     cout << "Enter the string \"hel
16     cin >> string1; // reads "hello" [space terminates input]
17
18     // output strings
19     cout << "string1 is: " << string1
20          << "\nstring2 is: " << string2;
21
22     cout << "\nstring1 with spaces between characters is:\n";
23
24     // output characters until null character is reached
25     for ( int i = 0; string1[ i ] != '\0'; i++ )
26        cout << string1[ i ] << ' ';
27
28     cin >> string1;  // reads "there"
29     cout << "\nstring1 is: " << string1 << endl;
30
31     return 0;  // indicates successful termination
32
33  } // end main
```

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

stringkeyboard.cpp

String and input from keyboard

```
1   // Fig. 4_12: fig04_12.cpp
2   // Treating character arrays as strings.
3   #include <iostream>
4
5   using std::cout;
6   using std::cin;
7   using std::endl;
8
9   int main()
10  {
11     char string1[ 20 ];                  // reserves 20 characters
12     char string2[] = "string literal"; // reserves 15 characters
13
14     // read string from user into array string2
15     cout << "Enter the string \"hello there\": ";
16     cin >> string1;  // reads "hello" [space terminates input]
17
18     // output strings
19     cout << "string1 is: " << string1
20          << "\nstring2 is: " << string2;
21
22     cout << "\nstring1 with spaces between characters is:\n";
23
24     // output characters until null character is reached
25     for ( int i = 0; string1[ i ] != '\0'; i++ )
26        cout << string1[ i ] << ' ';
27
28     cin >> string1;  // reads "there"
29     cout << "\nstring1 is: " << string1 << endl;
30
31     return 0;  // indicates successful termination
32
33  } // 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 );
```

    Variable names are 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

```
1   // Fig. 4.14: fig04_14.cpp
2   // Passing arrays and individual array elements to functions.
3   #include <iostream>
4
5   using std::cout;
6   using std::endl;
7
8   #include <iomanip>
9
10  using std::setw;
11
12  void modifyArray( int [], int );  // appears strange
13  void modifyElement( int );
14
15  int main()
16  {
17     const int arraySize = 5;                 // size of array a
18     int a[ arraySize ] = { 0, 1, 2, 3, 4 };  // initialize a
19
20     cout << "Effects of passing entire array by reference:"
21          << "\n\nThe values of the original array are:\n";
22
23     // output original array
24     for ( int i = 0; i < arraySize; i++ )
25        cout << setw( 3 ) << a[ i ];
```

Syntax for accepting an array in parameter list

arraytofunc.cpp

Passing Arrays to Functions

```
26
27       cout << endl;
28
29       // pass array a to modifyArray
30       modifyArray( a, arraySize );
31
32       cout << "The values of the modified array are:\n";
33
34       // output modified array
35       for ( int j = 0; j < arraySize; j++ )
36          cout << setw( 3 ) << a[ j ];
37
38       // output value of a[ 3 ]
39       cout << "\n\n\n"
40            << "Effects of passing array
41            << "\n\nThe value of a[3] is
42
43       // pass array element a[ 3 ] by va
44       modifyElement( a[ 3 ] );
45
46       // output value of a[ 3 ]
47       cout << "The value of a[3] is " << a[ 3 ] << endl;
48
49       return 0;  // indicates successful termination
50
51    } // 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

```
52
53    // in function modifyArray, "b" points to
54    // the original array "a" in memory
55    void modifyArray( int b[], int sizeOfArray )
56    {
57       // multiply each array element by 2
58       for ( int k = 0; k < sizeOfArray; k++ )
59          b[ k ] *= 2;
60
61    } // end function modifyArray
62
63    // in function modifyElement, "e" is a loca
64    // array element a[ 3 ] passed from main
65    void modifyElement( int e )
66    {
67       // multiply parameter by 2
68       cout << "Value in modifyElement is "
69            << ( e *= 2 ) << endl;
70
71    } // 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

```
1   // Fig. 4.15: fig04_15.cpp
2   // Demonstrating the const type qualifier.
3   #include <iostream>
4
5   using std::cout;
6   using std::endl;
7
8   void tryToModifyArray( const int [] );  // func
9
10  int main()
11  {
12     int a[] = { 10, 20, 30 };
13
14     tryToModifyArray( a );
15
16     cout << a[ 0 ] << ' ' << a[ 1 ] << ' ' << a[ 2 ] << '\n';
17
18     return 0;  // indicates successful termination
19
20  } // end main
21
22  // In function tryToModifyArray, "b" cannot be used
23  // to modify the original array "a" in main.
24  void tryToModifyArray( const int b[] )
25  {
26     b[ 0 ] /= 2;    // error
27     b[ 1 ] /= 2;    // error
28     b[ 2 ] /= 2;    // error
29
30  } // 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

Manipulating Arrays : Some algorithms examples

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

```
1     // Fig. 4.16: fig04_16.cpp
2     // This program sorts an array's values into ascending order.
3     #include <iostream>
4
5     using std::cout;
6     using std::endl;
7
8     #include <iomanip>
9
10    using std::setw;
11
12    int main()
13    {
14       const int arraySize = 10;  // size of array a
15       int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
16       int hold;  // temporary location used to swap array elements
17
18       cout << "Data items in original order\n";
19
20       // output original array
21       for ( int i = 0; i < arraySize; i++ )
22          cout << setw( 4 ) << a[ i ];
23
```

sortingarrays.cpp

Sorting Arrays

```
24       // bubble sort
25       // loop to control number of passes
26       for ( int pass = 0; pass < arraySize - 1; pass++ )
27
28          // loop to control number of comparisons per pass
29          for ( int j = 0; j < arraySize - 1; j++ )
30
31             // compare side-by-side elements and swap
32             // first element is greater than second e
33             if ( a[ j ] > a[ j + 1 ] ) {
34                hold = a[ j ];
35                a[ j ] = a[ j + 1 ];
36                a[ j + 1 ] = hold;
37
38             } // end if
39
```

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

```
40       cout << "\nData items in ascending order\n";
41
42       // output sorted array
43       for ( int k = 0; k < arraySize; k++ )
44          cout << setw( 4 ) << a[ k ];
45
46       cout << endl;
47
48       return 0;  // indicates successful termination
49
50    } // 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

```
1     // Fig. 4.17: fig04_17.cpp
2     // This program introduces the topic of survey data analysis.
3     // It computes the mean, median, and mode of the data.
4     #include <iostream>
5
6     using std::cout;
7     using std::endl;
8     using std::fixed;
9     using std::showpoint;
10
11    #include <iomanip>
12
13    using std::setw;
14    using std::setprecision;
15
16    void mean( const int [], int );
17    void median( int [], int );
18    void mode( int [], int [], int );
19    void bubbleSort( int[], int );
20    void printArray( const int[], int );
21
22    int main()
23    {
24       const int responseSize = 99;  // size of array responses
25
```

MeanMedianMode.cpp

Mean, Median and Mode Using Arrays

```
26       int frequency[ 10 ] = { 0 };  // initialize array frequency
27
28       // initialize array responses
29       int response[ responseSize ] =
30              { 6, 7, 8, 9, 8, 7, 8, 9, 8, 9,
31                7, 8, 9, 5, 9, 8, 7, 8, 7, 8,
32                6, 7, 8, 9, 3, 9, 8, 7, 8, 7,
33                7, 8, 9, 8, 9, 8, 9, 7, 8, 9,
34                6, 7, 8, 7, 8, 7, 9, 8, 9, 2,
35                7, 8, 9, 8, 9, 8, 9, 7, 5, 3,
36                5, 6, 7, 2, 5, 3, 9, 4, 6, 4,
37                7, 8, 9, 6, 8, 7, 8, 9, 7, 8,
38                7, 4, 4, 2, 5, 3, 8, 7, 5, 6,
39                4, 5, 6, 1, 6, 5, 7, 8, 7 };
40
41       // process responses
42       mean( response, responseSize );
43       median( response, responseSize );
44       mode( frequency, response, responseSize );
45
46       return 0;  // indicates successful termination
47
48    } // end main
49
```

MeanMedianMode.cpp

Mean, Median and Mode Using Arrays

```
50    // calculate average of all response values
51    void mean( const int answer[], int arraySize )
52    {
53       int total = 0;
54
55       cout << "********\n  Mean\n********\n";
56
57       // total response values
58       for ( int i = 0; i < arraySize; i++ )
59          total += answer[ i ];
60
61       // format and output results
62       cout << fixed << setprecision( 4 );
63
64       cout << "The mean is the average value of the data\n"
65            << "items. The mean is equal to the total of\n"
66            << "all the data items divided by the number\n"
67            << "of data items (" << arraySize
68            << "). The mean value for\nthis run is: "
69            << total << " / " << arraySize << " = "
70            << static_cast< double >( total ) / arraySize
71            << "\n\n";
72
73    } // end function mean
74
```

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

MeanMedianMode.cpp

Mean, Median and Mode Using Arrays

```
75    // sort array and determine median element's value
76    void median( int answer[], int size )
77    {
78       cout << "\n********\n Median\n********\n"
79            << "The unsorted array of responses is";
80
81       printArray( answer, size );  // output unsorted array
82
83       bubbleSort( answer, size );  // sort array
84
85       cout << "\n\nThe sorted array is";
86       printArray( answer, size );  // output sorted array
87
88       // display median element
89       cout << "\n\nThe median is element " << size / 2
90            << " of\nthe sorted " << size
91            << " element array.\nFor this run the median is "
92            << answer[ size / 2 ] << "\n\n";
93
94    } // end function median
95
```

MeanMedianMode.cpp

Mean, Median and Mode Using Arrays

```
96    // determine most frequent response
97    void mode( int freq[], int answer[], int size )
98    {
99       int largest = 0;    // represents largest frequency
100     int modeValue = 0;  // represents most frequent response
101
102     cout << "\n********\n  Mode\n********\n";
103
104     // initialize frequencies to 0
105     for ( int i = 1; i <= 9; i++ )
106        freq[ i ] = 0;
107
108     // summarize frequencies
109     for ( int j = 0; j < size; j++ )
110        ++freq[ answer[ j ] ];
111
112     // output headers for result columns
113     cout << "Response" << setw( 11 ) << "Frequency"
114          << setw( 19 ) << "Histogram\n\n" << setw( 55 )
115          << "1    1    2    2\n" << setw( 56 )
116          << "5    0    5    0    5\n\n";
117
```

MeanMedianMode.cpp

Mean, Median and Mode Using Arrays

```
118     // output results
119     for ( int rating = 1; rating <= 9; rating++ ) {
120        cout << setw( 8 ) << rating << setw( 11 )
121             << freq[ rating ] << "          ";
122
123        // keep track of mode value and largest frequ
124        if ( freq[ rating ] > largest ) {
125           largest = freq[ rating ];
126           modeValue = rating;
127
128        } // end if
129
130        // output histogram bar representing frequency value
131        for ( int k = 1; k <= freq[ rating ]; k++ )
132           cout << '*';
133
134        cout << '\n';  // begin new line of output
135
136     } // end outer for
137
138     // display the mode value
139     cout << "The mode is the most frequent value.\n"
140          << "For this run the mode is " << modeValue
141          << " which occurred " << largest << " times." << endl;
142
143  } // 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

```
144
145 // function that sorts an array with bubble sort algorithm
146 void bubbleSort( int a[], int size )
147 {
148    int hold;  // temporary location used to swap elements
149
150    // loop to control number of passes
151    for ( int pass = 1; pass < size; pass++ )
152
153       // loop to control number of comparisons per pass
154       for ( int j = 0; j < size - 1; j++ )
155
156          // swap elements if out of order
157          if ( a[ j ] > a[ j + 1 ] ) {
158             hold = a[ j ];
159             a[ j ] = a[ j + 1 ];
160             a[ j + 1 ] = hold;
161
162          } // end if
163
164 } // end function bubbleSort
165
```

MeanMedianMode.cpp

Mean, Median and Mode Using Arrays

```
166  // output array contents (20 values per row)
167  void printArray( const int a[], int size )
168  {
169     for ( int i = 0; i < size; i++ ) {
170
171        if ( i % 20 == 0 )  // begin new line every 20 values
172           cout << endl;
173
174        cout << setw( 2 ) << a[ i ];
175
176     } // end for
177
178  } // 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 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

```
1   // Fig. 4.19: fig04_19.cpp
2   // Linear search of an array.
3   #include <iostream>
4
5   using std::cout;
6   using std::cin;
7   using std::endl;
8
9   int linearSearch( const int [], int, int );  // prototype
10
11  int main()
12  {
13     const int arraySize = 100;  // size of array a
14     int a[ arraySize ];         // create array a
15     int searchKey;              // value to locate in a
16
17     for ( int i = 0; i < arraySize; i++ )  // create some data
18        a[ i ] = 2 * i;
19
20     cout << "Enter integer search key: ";
21     cin >> searchKey;
22
23     // attempt to locate searchKey in array a
24     int element = linearSearch( a, searchKey, arraySize );
25
```

Takes array, search key, and array size.

linearSearch.cpp

Linear Search

```
26       // display results
27       if ( element != -1 )
28          cout << "Found value in element " << element << endl;
29       else
30          cout << "Value not found" << endl;
31
32       return 0;  // indicates successful termination
33
34    } // end main
35
36    // compare key to every element of array until location is
37    // found or until end of array is reached; return subscript of
38    // element if key or -1 if key not found
39    int linearSearch( const int array[], int key, int sizeOfArray )
40    {
41       for ( int j = 0; j < sizeOfArray; j++ )
42
43          if ( array[ j ] == key )  // if found,
44             return j;              // return location of key
45
46       return -1;  // key not found
47
48    } // end function linearSearch
```

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

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

Binary Search

```
1      // Fig. 4.20: fig04_20.cpp
2      // Binary search of an array.
3      #include <iostream>
4
5      using std::cout;
6      using std::cin;
7      using std::endl;
8
9      #include <iomanip>
10
11    using std::setw;
12
13    // function prototypes
14    int binarySearch( const int [], int, int, int, int );
15    void printHeader( int );
16    void printRow( const int [], int, int, int, int );
17
18    int main()
19    {
20       const int arraySize = 15;  // size of array a
21       int a[ arraySize ];        // create array a
22       int key;                   // value to locate in a
23
24       for ( int i = 0; i < arraySize; i++ )  // create some data
25          a[ i ] = 2 * i;
26
```

binarySearch.cpp

Binary Search

```
27       cout << "Enter a number between 0 and 28: ";
28       cin >> key;
29
30       printHeader( arraySize );
31
32       // search for key in array a
33       int result =
34          binarySearch( a, key, 0, arraySize - 1, arraySize );
35
36       // display results
37       if ( result != -1 )
38          cout << '\n' << key << " found in array element "
39               << result << endl;
40       else
41          cout << '\n' << key << " not found" << endl;
42
43       return 0;  // indicates successful termination
44
45    } // end main
46
```

binarySearch.cpp

Binary Search

```
47    // function to perform binary search of an array
48    int binarySearch( const int b[], int searchKey, int low,
49       int high, int size )
50    {
51       int middle;
52
53       // loop until low subscript is greate
54       while ( low <= high ) {
55
56          // determine middle element of subarray being searched
57          middle = ( low + high ) / 2;
58
59          // display subarray used in this loop iteration
60          printRow( b, low, middle, high, size );
61
```

Determine middle element

binarySearch.cpp

Binary Search

```
62          // if searchKey matches middle element, return middle
63          if ( searchKey == b[ middle ] )  // match
64             return middle;
65
66          else
67
68             // if searchKey less than middle elem
69             // set new high element
70             if ( searchKey < b[ middle ] )
71                high = middle - 1;  // search low end of array
72
73             // if searchKey greater than middl
74             // set new low element
75             else
76                low = middle + 1;    // search h
77       }
78
79       return -1;  // searchKey not found
80
81    } // 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

```
82
83    // print header for output
84    void printHeader( int size )
85    {
86       cout << "\nSubscripts:\n";
87
88       // output column heads
89       for ( int j = 0; j < size; j++ )
90          cout << setw( 3 ) << j << ' ';
91
92       cout << '\n';  // start new line of output
93
94       // output line of - characters
95       for ( int k = 1; k <= 4 * size; k++ )
96          cout << '-';
97
98       cout << endl;  // start new line of output
99
100   } // end function printHeader
101
```

binarySearch.cpp

Binary Search

```
102 // print one row of output showing the current
103 // part of the array being processed
104 void printRow( const int b[], int low, int mid,
105    int high, int size )
106 {
107    // loop through entire array
108    for ( int m = 0; m < size; m++ )
109
110       // display spaces if outside current subarray range
111       if ( m < low || m > high )
112          cout << "    ";
113
114       // display middle element marked with a *
115       else
116
117          if ( m == mid )           // mark middle value
118             cout << setw( 3 ) << b[ m ] << '*';
119
120          // display other elements in subarray
121          else
122             cout << setw( 3 ) << b[ m ] << ' ';
123
124    cout << endl;  // start new line of output
125
126 } // 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

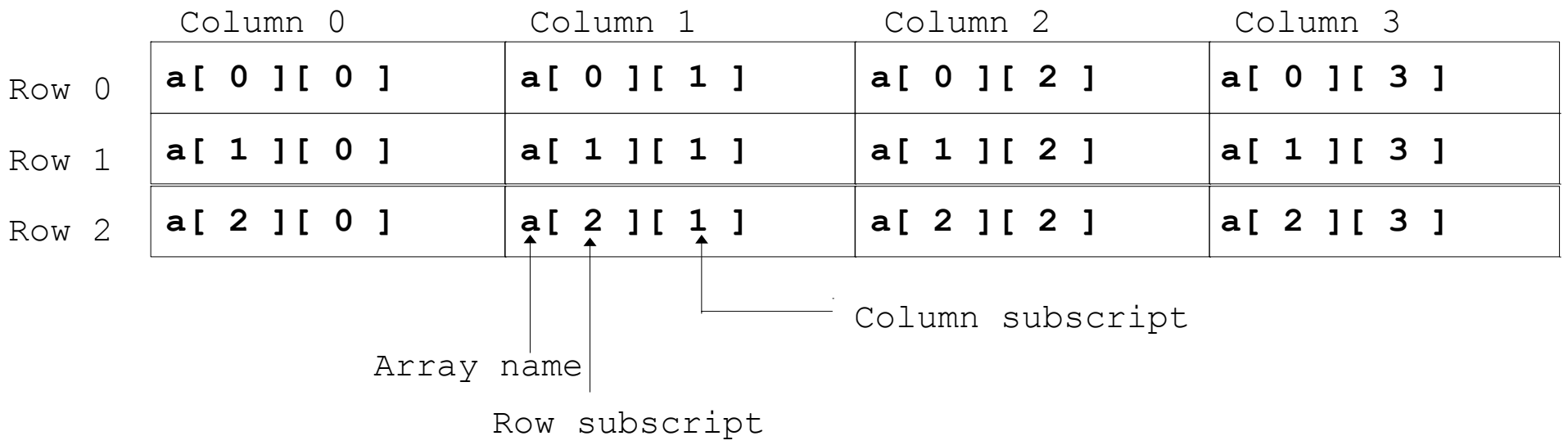
Binary Search

```
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



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

- 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 ] );
```

1	0
3	4

Multiple-Subscripted Arrays

```
1   // Fig. 4.22: fig04_22.cpp
2   // Initializing multidimensional arrays.
3   #include <iostream>
4
5   using std::cout;
6   using std::endl;
7
8   void printArray( int [][ 3 ] );
9
10  int main()
11  {
12     int array1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } };
13     int array2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 };
14     int array3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };
15
16     cout << "Values in array1 by row are:" << endl;
17     printArray( array1 );
18
19     cout << "Values in array2 by row are:" << endl;
20     printArray( array2 );
21
22     cout << "Values in array3 by row are:" << endl;
23     printArray( array3 );
24
25     return 0;  // indicates successful termination
26
27  } // 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

```
28
29    // function to output array with two rows
30    void printArray( int a[][ 3 ] )
31    {
32       for ( int i = 0; i < 2; i++ ) {    // f
33
34          for ( int j = 0; j < 3; j++ )   // output column values
35             cout << a[ i ][ j ] << ' ';
36
37          cout << endl;  // start new line of output
38
39       } // end outer for structure
40
41    } // 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

```
1   // Fig. 4.23: fig04_23.cpp
2   // Double-subscripted array example.
3   #include <iostream>
4
5   using std::cout;
6   using std::endl;
7   using std::fixed;
8   using std::left;
9
10  #include <iomanip>
11
12  using std::setw;
13  using std::setprecision;
14
15  const int students = 3;   // number of students
16  const int exams = 4;      // number of exams
17
18  // function prototypes
19  int minimum( int [][ exams ], int, int );
20  int maximum( int [][ exams ], int, int );
21  double average( int [], int );
22  void printArray( int [][ exams ], int, int );
23
```

QuizArray.cpp

Multiple-Subscripted Arrays

```
24    int main()
25    {
26       // initialize student grades for three students (rows)
27       int studentGrades[ students ][ exams ] =
28          { { 77, 68, 86, 73 },
29            { 96, 87, 89, 78 },
30            { 70, 90, 86, 81 } };
31
32       // output array studentGrades
33       cout << "The array is:\n";
34       printArray( studentGrades, students, exams );
35
36       // determine smallest and largest grade values
37       cout << "\n\nLowest grade: "
38            << minimum( studentGrades, students, exams )
39            << "\nHighest grade: "
40            << maximum( studentGrades, students, exams ) << '\n';
41
42       cout << fixed << setprecision( 2 );
43
```

QuizArray.cpp

Multiple-Subscripted Arrays

```
44       // calculate average grade for each student
45       for ( int person = 0; person < students; person++ )
46          cout << "The average grade for student " << person
47               << " is "
48               << average( studentGrades[ person ], exams )
49               << endl;
50
51       return 0;  // indicates successful termination
52
53    } // end main
54
55    // find minimum grade
56    int minimum( int grades[][ exams ], int pupils, int tests )
57    {
58       int lowGrade = 100; // initialize to highest possible grade
59
60       for ( int i = 0; i < pupils; i++ )
61
62          for ( int j = 0; j < tests; j++ )
63
64             if ( grades[ i ][ j ] < lowGrade )
65                lowGrade = grades[ i ][ j ];
66
67       return lowGrade;
68
69    } // 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

```
70
71    // find maximum grade
72    int maximum( int grades[][ exams ], int pupils, int tests )
73    {
74       int highGrade = 0;  // initialize to lowest possible grade
75
76       for ( int i = 0; i < pupils; i++ )
77
78          for ( int j = 0; j < tests; j++ )
79
80             if ( grades[ i ][ j ] > highGrade )
81                highGrade = grades[ i ][ j ];
82
83       return highGrade;
84
85    } // end function maximum
86
87    // determine average grade for particular student
88    double average( int setOfGrades[], int tests )
89    {
90       int total = 0;
91
92       // total all grades for one student
93       for ( int i = 0; i < tests; i++ )
94          total += setOfGrades[ i ];
95
96       return static_cast< double >( total ) / tests;  // average
97
98    } // end function maximum
```

QuizArray.cpp

Multiple-Subscripted Arrays

```
99
100 // Print the array
101 void printArray( int grades[][ exams ], int pupils, int tests )
102 {
103    // set left justification and output column heads
104    cout << left << "                [0]  [1]  [2]  [3]";
105
106    // output grades in tabular format
107    for ( int i = 0; i < pupils; i++ ) {
108
109       // output label for row
110       cout << "\nstudentGrades[" << i << "] ";
111
112       // output one grades for one student
113       for ( int j = 0; j < tests; j++ )
114          cout << setw( 5 ) << grades[ i ][ j ];
115
116    } // end outer for
117
118 } // 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)Write and test a function that returns the index of the minimum value among the first n elements of a given array (MinInArray.cpp):

```
float min(float a[], int n);
```

2)Write and test the following function:

```
double stddev(double x[], int n);
```

The function returns the standard deviation of a dataset of `n` numbers x_0, ..., x_{n-1} defined by the formula (StandardDeviation.cpp)

$$s = \sqrt{\frac{\sum_{i=0}^{n-1} (x_i - \bar{x})^2}{n-1}}$$

Esercitazione 5

3) 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

4) 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
```

5) 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
```