LECTURE 8

Functions

- **Functions**
  - Modularize a program
  - Software reusability
    - 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 call

- We already discussed the function call previously
- We are discussing it again here for revision purpose

- Function is invoked by function call
- A function call specifies the function name and provides information (as arguments) that the called function needs
- Functions called by writing
  - **functionName (argument);**

  or
  - **functionName(argument1, argument2, …);**

- The argument can be a constant, variable or expression

Function Definitions

- Each program we observed so far consisted of a function called **main**
- Function main called standard library functions to accomplish its task
- Now we are going to discuss how programmer write there own customized functions
- For programmer defined functions the important components are
  - Function definition
  - Function prototype

Function Definition

- Describes how the function does its task
- Can appear before or after the function is called
- **Format**

  return-value-type function-name **(** ***parameter-list*** **)**

  {

  declarations and statements

  }

  - **Function-name**
    - is any valid identifier e.g. square
  - **Return-value-type**
    - is the data type of the result returned from the function to the caller
    - Return value type **void** indicated that the function does not return a value
  - **Parameter-list**
    - Is a comma-separated list of the arguments
    - containing the declarations (data type needed for each argument) of the parameters received by the function when it is called
    - If the function does not receive any values, **parameter-list** is **void** or simply left **empty**

Function Definition

- Example function

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

- **`return`** keyword
  - **Format** ***return expression;***
  - Returns the value calculated by the function
  - 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*

Function Prototype

- Must appear in the code before the function can be called
- The compiler use the function prototypes to validate function call
- **Format /syntax**
  - ***Return-value-type Function_Name(Parameter_List);***

- Function prototype tells the compiler
  - **The function's name**
  - **Type of data returned by the function (**void if return nothing**)**
  - **Number of parameters the function accepts to receive**
  - **Types of parameters**
  - **The order in which these parameters are expected**
- Function prototype is not required if the function definition appears before the first use function's first use in the program.
- In such case the function definition also acts as the function prototype

Example program

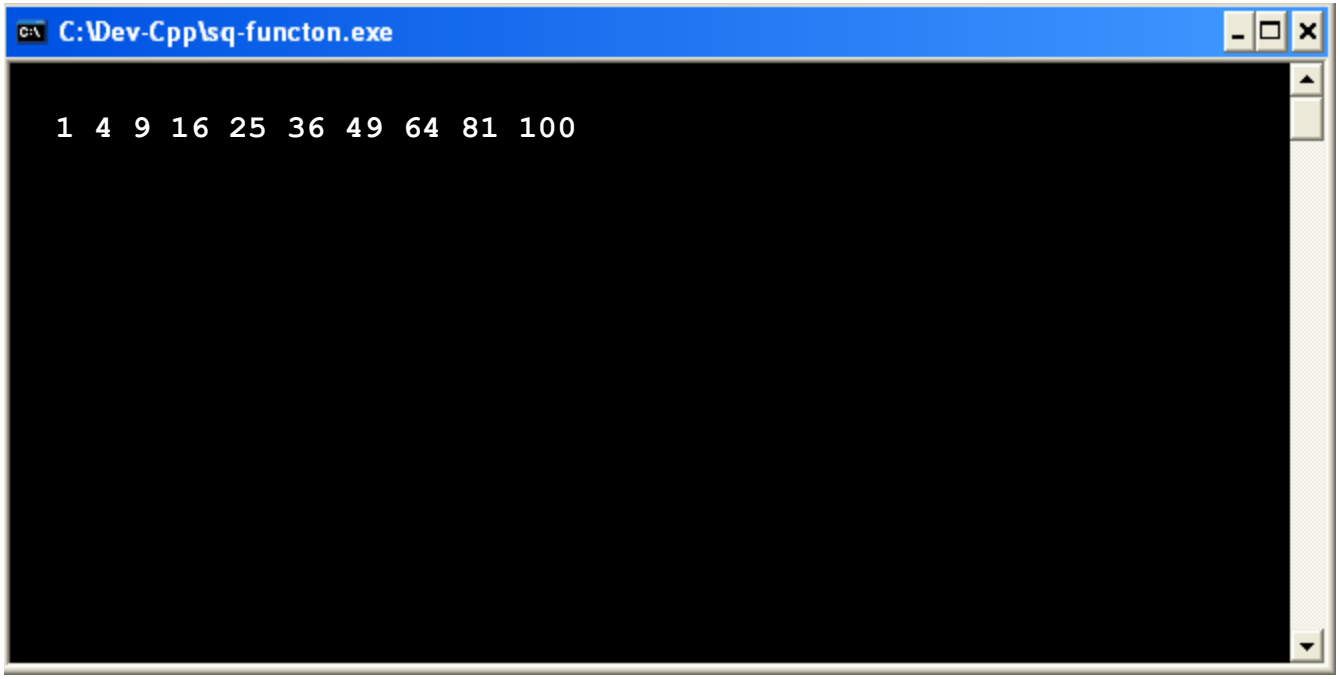
```
1   // program calculate the square of integers 1 to 10
2   // Creating and using a programmer-defined function
3   #include <iostream>
4
5   using std::cout;
6   using std::endl;
7
8   int square( int );   // function prototype
9
10  int main()
11  {
12    // loop 10 times and calculate and output
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
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
```

Function prototype: specifies data types of arguments and return values. **`square`** expects and **`int`**, and returns an **`int`**

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

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

Output



Function prototype

- Prototype must match function definition
  - Function prototype

```
double maximum( double, double, double );
```

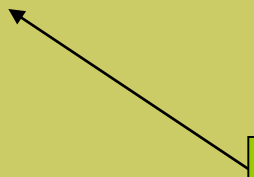
  - Definition

```
double maximum( double x, double y, double z)
{
  …
}
```

- observe this in the following program example
- This example program uses a programmer-defined function maximum to determine and return the largest of three integers

Example program

```
1   // using programmer defined function
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
```

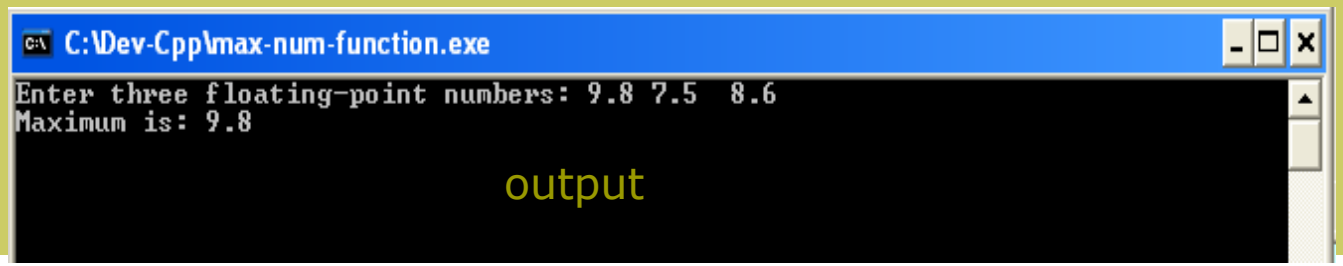


Function **`maximum`** takes 3 arguments (all **`double`**) and returns a **`double`**.

Example program

```
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.



Function signature

- Function signature is also called simply signature
- It is the portion of a function prototype that includes
  - name of function
  - and parameters (types of its argument)
- **Example**

```
double maximum( double, double, double );
```

Function signature

Argument correction

- Forcing the argument to the appropriate type
- Argument values that do not correspond precisely to the parameter types in the function prototype are converted to proper type before the function is called
- **Example**
- **sqrt** a math library function has prototype in <**cmath**> that specifies a **double** argument
- **sqrt** can be called with an integer argument, the function will still work correctly
  - Statement `cout << sqrt(4)` correctly evaluate and prints `2`
  - Function prototype causes compiler to convert **int** value **4** to the **double** value **4.0** before the value is passed to **sqrt**
  - However changing from double to int can truncate data e.g. 3.5 to 3
  - Converting values to lower type can lead to incorrect result

Argument correction

- As we saw some of the conversions may lead to incorrect results if proper promotion rules are not followed

- **Promotion rules**
  - The promotion rules specify how types can be converted to other types without losing data
  - Promotion rules apply to expressions containing values of two or more data types ***(mixed-type-expressions)***
  - type of each value in a mixed-type expression is promoted to the "**highest**" type in the expression
  - Also used when the type of an argument to a function does not match the parameter type specified in the function definition
  - Converting values to lower type can result in incorrect values

Promotion hierarchy for built in data types

Data types	
`long double`	
`double`	
`float`	
`unsigned long int`	(synonymous with `unsigned long`)
`long int`	(synonymous with `long`)
`unsigned int`	(synonymous with `unsigned`)
`int`	
`unsigned short int`	(synonymous with `unsigned short`)
`short int`	(synonymous with `short`)
`unsigned char`	
`char`	
`bool`	(`false` becomes 0, `true` becomes 1)

Type casting

- A Type Cast produces the value of one type from another type
- Converting values to lower type can result in incorrect values
- A value can be converted to a lower type only by explicitly assigning the value to a variable of lower type
- Cast operator is used for this purpose

- **Example**
  - **static_cast<double>(total) //total is of type integer**
  - produces a **double** representing the **integer** value of **total** (operand in parenthesis)
  - **double** is higher type and **int** is lower type

Header Files

- **Library header file**
  - Contain function prototypes for library functions
  - Definition of various data type and constants needed by library functoins
  - `<cstdlib>` , `<cmath>`, etc.
  - Load with `#include <filename>`
    - Example:

```
#include <cmath>
```

- **Custom header files**
  - Defined by the programmer
  - Save as `filename.h`
  - Included in a program using **#include** preprocessor directive
  - `#include "filename.h"`
  - Example
    - `#include "square.h"` //square.h is programmer defined header file

- The programmer defined header file should end with .h

Random Number Generation

- Function prototype of rand function can be found in <**cstdlib**> header file
- Statement `i = rand()`
- The function rand generates an unsigned integer between **0** and **`RAND-MAX`**
  - RAND-MAX is a symbolic constant defined in <**cstdlib**> header file
  - Maximum value is 32767-maximum possible value for 2 byte integer
- If **rand** truly produce an integer at random, every number between **`zero`** and **`RAND-MAX`** has an equal probability of being chosen each time **`rand`** is called

Random Number Generation

- **Example program**
  - To demonstrate rand consider a program to simulate 20 rolls of a six sided die and print the value of each roll.

- **Scaling and shifting**
  - To produce integer in range 0 to 5 use modulus operator (%) with rand
  - Modulus (remainder) operator: `%`
    - `10 % 3` is `1`
    - `x % y` is between `0` and `y - 1`

Random Number Generation

- **General scaling and shifting**
  - Number = shiftingValue + `rand() %` scalingFactor
  - shiftingValue = first number in desired range
  - scalingFactor = width of desired range

  - **Example**

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

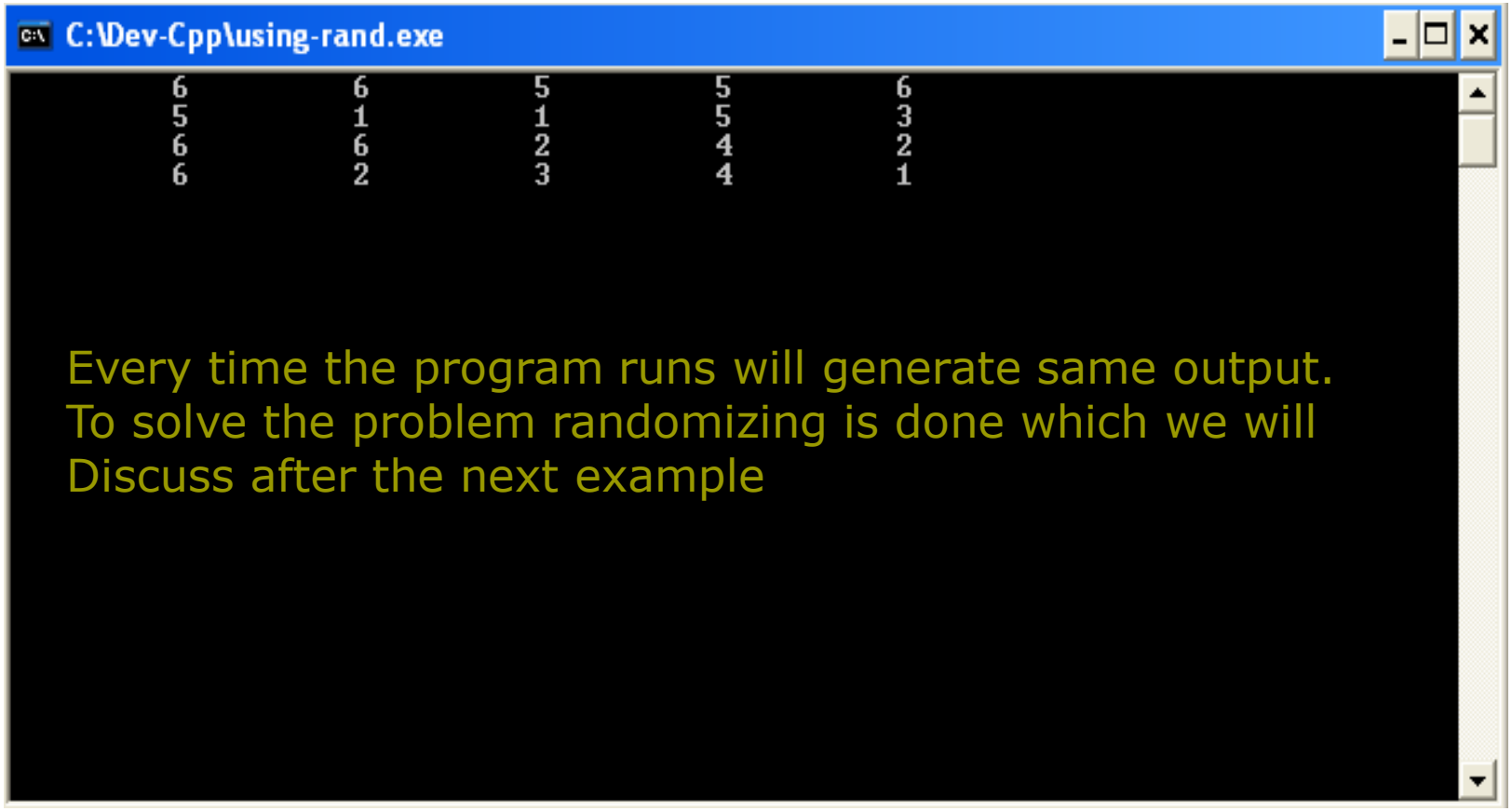
    - "`Rand() % 6`" generates a number between `0` and `5` (scaling)
    - "`+ 1`" makes the range 1 to 6 (shift)
    - **6** is the scaling factor
  - Next: program to roll dice

C++ code

```
1   // rolling die
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.

Output


C:\Dev-Cpp\using-rand.exe
6 6 5 5 6
5 1 1 5 3
6 6 2 4 2
6 2 3 4 1
Every time the program runs will generate same output.
To solve the problem randomizing is done which we will
Discuss after the next example

Random Number Generation

- **Example program**
  - **Program statement**
    - Program to show distribution of **`rand()`**
    - Simulate 6000 rolls of a die
    - Print the count for number of 1's, 2's, 3's, etc. rolled
    - Should be roughly 1000 of each

C++ code

```
1   // Fig. 3.8: fig03_08.cpp
2   // Roll a six-sided die 6000 times.
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     int frequency1 = 0;
17     int frequency2 = 0;
18     int frequency3 = 0;
19     int frequency4 = 0;
20     int frequency5 = 0;
21     int frequency6 = 0;
22     int face;  // represents one roll of the die
23
```

C++ code

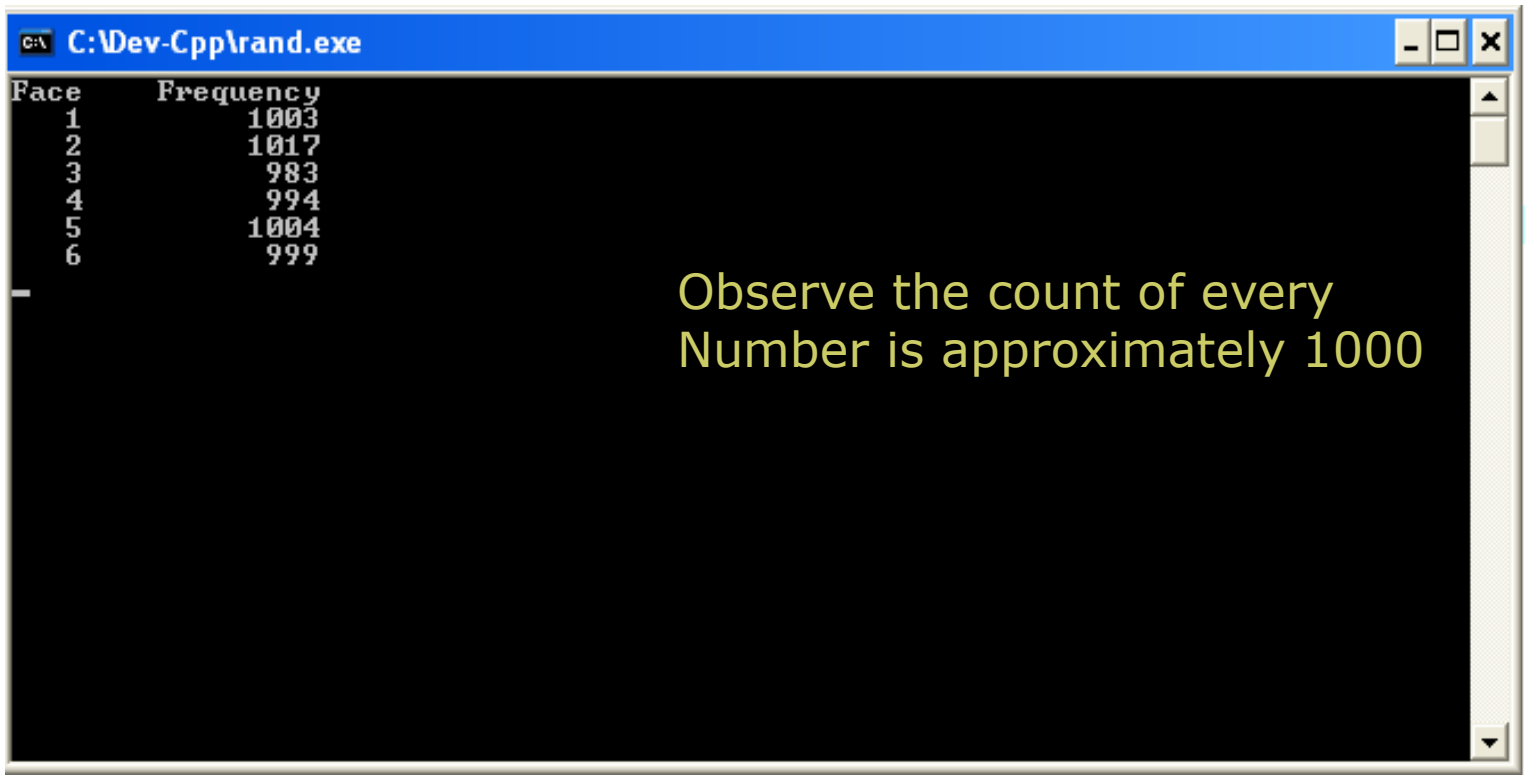
```
24    // loop 6000 times and summarize results
25    for ( int roll = 1; roll <= 6000; roll++ ) {
26       face = 1 + rand() % 6;  // random number from 1 to 6
27
28       // determine face value and increment appropriate counter
29       switch ( face ) {
30
31          case 1:         // rolled 1
32             ++frequency1;
33             break;
34
35          case 2:         // rolled 2
36             ++frequency2;
37             break;
38
39          case 3:         // rolled 3
40             ++frequency3;
41             break;
42
43          case 4:         // rolled 4
44             ++frequency4;
45             break;
46
47          case 5:         // rolled 5
48             ++frequency5;
49             break;
```

C++ code

```
50
51          case 6:         // rolled 6
52             ++frequency6;
53             break;
54
55          default:        // invalid value
56             cout << "Program should never get here!";
57
58       } // end switch
59
60    } // end for
61
62    // display results in tabular fo
63    cout << "Face" << setw( 13 ) << "Frequency"
64         << "\n   1" << setw( 13 ) << frequency1
65         << "\n   2" << setw( 13 ) << frequency2
66         << "\n   3" << setw( 13 ) << frequency3
67         << "\n   4" << setw( 13 ) << frequency4
68         << "\n   5" << setw( 13 ) << frequency5
69         << "\n   6" << setw( 13 ) << frequency6 << endl;
70
71    return 0;  // indicates successful termination
72
73 } // end main
```

Default case included even though it should never be reached. This is a matter of good coding style

Output


C:\Dev-Cpp\rand.exe
Face Frequency
1 1003
2 1017
3 983
4 994
5 1004
6 999
Observe the count of every
Number is approximately 1000

Random Number Generation

- **Calling rand() repeatedly**
  - Gives the same sequence of numbers each time program is executed
- **Pseudorandom numbers**
  - Preset sequence of "random" numbers
  - Same sequence generated whenever program run
- **To get different random sequences**
  - **Provide a seed value**
    - Like a random starting point in the sequence
    - The same seed will give the same sequence
    - The standard library function **srand** is used for randomizing
  - `srand(seed);`
    - **<cstdlib>**
    - Used before `rand()` to set the seed

Reading assignment

- C++ how to program by Deitel and Deitel
- Topic 3.8 Random number generation
- Randomizing die Roll program
- fig 3.9 (3^{rd}/4^{th} edition)

- Review the program to understand the use of **srand (seed);** and

- How it produces different sequence of random numbers for each execution of the program

- What is the use of **srand (time(0));**