# 403-2:Functions

Mojtaba kordabadi

# Introduction

- Divide and conquer
  - Construct a program from smaller pieces or components
  - Each piece more manageable than the original program

# 3.2 Program Components in C++

- Programs written by
  - combining new functions with “prepackaged” functions in the C++ standard library.
  - new classes with “prepackaged” classes.
- The standard library provides a rich collection of functions.
- Functions are invoked by a function call
  - A function call specifies the function name and provides information (as arguments) that the called function needs
  - Boss to worker analogy:

    *A boss (the calling function or caller) asks a worker (the called function) to perform a task and return (i.e., report back) the results when the task is done.*

# Program Components in C++

- Function definitions
  - Only written once
  - These statements are hidden from other functions.
  - Boss to worker analogy:

    *The boss does not know how the worker gets the job done; he just wants it done*

# 3.3 Math Library Functions

- Math library functions
  - Allow the programmer to perform common mathematical calculations
  - Are used by including the header file **<cmath>**
- Functions called by writing

  *functionName* (*argument*)
- Example

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

  - Calls the **sqrt** (square root) function. The preceding statement would print **30**
  - The **sqrt** function takes an argument of type **double** and returns a result of type **double**, as do all functions in the math library

# 3.3 Math Library Functions

- Function arguments can be
  - Constants

```
sqrt( 4 );
```

  - Variables

```
sqrt( x );
```

  - Expressions

```
sqrt( sqrt( x ) );
sqrt( 3 - 6x );
```

# 3.4 Functions

- Functions
  - Allow the programmer to modularize a program
- 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 when the function is called that provide the function with outside information

# Functions

- Why write functions?
  - modularity
  - re-use
  - maintenance / testing

# 3.5 Function Definitions

- Create customized functions to
  - Take in data
  - Perform operations
  - Return the result
- Format for function definition:

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

- Example:

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

```
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    for ( int x = 1; x <= 10; x++ )
13       cout << square( x ) << "  ";
14
15    cout << endl;
16    return 0;
17 }
18
19 // Function definition
20 int square( int y )
21 {
22    return y * y;
23 }
```

Notice how parameters and return value are declared.

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

```cpp
// Fig. 3.4: fig03_04.cpp
// Finding the maximum of three integers
#include <iostream>

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

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

int main()
{
   int a, b, c;

   cout << "Enter three integers: ";
   cin >> a >> b >> c;

   // a, b and c below are arguments to
   // the maximum function call
   cout << "Maximum is: " << maximum( a, b, c ) << endl;
```

```
21
22    return 0;
23 }
24
25 // Function maximum definition
26 // x, y and z below are parameters to
27 // the maximum function definition
28 int maximum( int x, int y, int z )
29 {
30    int max = x;
31
32    if ( y > max )
33       max = y;
34
35    if ( z > max )
36       max = z;
37
38    return max;
39 }
```

```
Enter three integers: 22 85 17
Maximum is: 85

Enter three integers: 92 35 14
Maximum is: 92

Enter three integers: 45 19 98
Maximum is: 98
```

# 3.6 Function Prototypes

- Function prototype
  - Function name
  - Parameters
    - Information the function takes in
    - C++ is “strongly typed” – error to pass a parameter of the wrong type
  - Return type
    - Type of information the function passes back to caller (default **`int`**)
    - **`void`** signifies the function returns nothing
  - Only needed if function definition comes after the function call in the program
- Example:

```
int maximum( int, int, int );
```

  - Takes in 3 **`int`**s
  - Returns an **`int`**

# 3.7 Header Files

- Header files
  - Contain function prototypes for library functions
  - **<cstdlib>** , **<cmath>**, etc.
  - Load with **#include <filename>**
    - Example:

```
#include <cmath>
```

- Custom header files
  - Defined by the programmer
  - Save as **filename.h**
  - Loaded into program using

```
#include "filename.h"
```

# 3.8 Random Number Generation

- **`rand`** function

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

  - Load **`<cstdlib>`**
  - Generates a pseudorandom number between **`0`** and **`RAND_MAX`** (usually 32767)
    - A pseudorandom number is a preset sequence of "random" numbers
    - The same sequence is generated upon every program execution
- **`srand`** function
  - Jumps to a seeded location in a "random" sequence

```
srand( seed );
srand( time( 0 ) );   //must include <ctime>
```

  - **`time( 0 )`**
    - The time at which the program was compiled
  - Changes the seed every time the program is compiled, thereby allowing **`rand`** to generate random numbers

# 3.8 Random Number Generation

- Scaling
  - Reduces random number to a certain range
  - Modulus ( `%` ) operator
    - Reduces number between 0 and **`RAND_MAX`** to a number between 0 and the scaling factor
  - Example

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

    - Generates a number between **`1`** and **`6`**

```
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>
13
14 int main()
15 {
16    for ( int i = 1; i <= 20; i++ ) {
17       cout << setw( 10 ) << ( 1 + rand() % 6 );
18
19       if ( i % 5 == 0 )
20          cout << endl;
21    }
22
23    return 0;
24 }
```

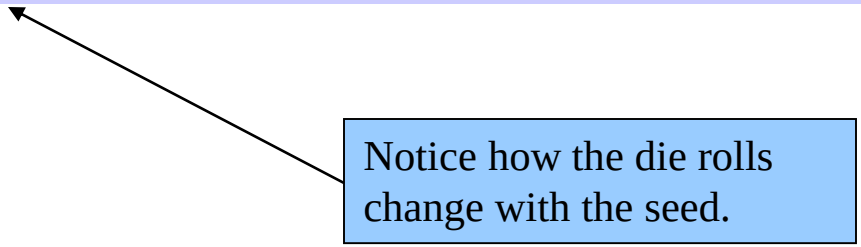
Notice **`rand() % 6.`** This returns a number between **0** and 5 (scaling). Add 1 to get a number between **1** and **6.**

Executing the program again gives the same "random" dice rolls.

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

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

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

#include <iomanip>

using std::setw;

#include <cstdlib>

int main()
{
   unsigned seed;

   cout << "Enter seed: ";
   cin >> seed;
   srand( seed );

   for ( int i = 1; i <= 10; i++ ) {
      cout << setw( 10 ) << 1 + rand() % 6;

      if ( i % 5 == 0 )
         cout << endl;
   }

   return 0;
}
```

```
Enter seed: 67
        1       6       5       1       4
        5       6       3       1       2

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

Enter seed: 67
        1       6       5       1       4
        5       6       3       1       2
```




# Program Output

# 3.9 Storage Classes

- Storage class specifiers
  - Storage class
    - Where object exists in memory
  - Scope
    - Where object is referenced in program
  - Linkage
    - Where an identifier is known
- Automatic storage
  - Object created and destroyed within its block
  - **`auto`**
    - Default for local variables.
    - Example:

```
auto float x, y;
```

  - **`register`**
    - Tries to put variables into high-speed registers
  - Can only be used with local variables and parameters

# Storage Classes

- Static storage
  - Variables exist for entire program execution
  - **`static`**
    - Local variables defined in functions
    - Keep value after function ends
    - Only known in their own function
  - **`Extern`**
    - Default for global variables and functions.
    - Known in any function

# Scope Rules

- File scope
  - Defined outside a function, known in all functions
  - Examples include, global variables, function definitions and functions prototypes
- Function scope
  - Can only be referenced inside a function body
  - Only labels (**`start:`**, **`case:`**, etc.)
- Block scope
  - Declared inside a block.  Begins at declaration, ends at }
  - Variables, function parameters (local variables of function)
  - Outer blocks “hidden” from inner blocks if same variable name
- Function prototype scope
  - Identifiers in parameter list
  - Names in function prototype optional, and can be used anywhere

```
1  // Fig. 3.12: fig03 12.cpp
2  // A scoping example
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  void a( void );   // function prototype
9  void b( void );   // function prototype
10 void c( void );   // function prototype
11
12 int x = 1;       // global var
13
14 int main()
15 {
16    int x = 5;    // local variable to main
17
18    cout << "local x in outer scope of main is " << x << endl;
19
20    {             // start new scope
21       int x = 7;
22
23       cout << "local x in inner scope of main is " << x << endl;
24    }             // end new scope
25
26    cout << "local x in outer scope of main is " << x << endl;
27
28    a();          // a has automatic local x
29    b();          // b has static loca
30    c();          // c uses global x
31    a();          // a reinitializes a
32    b();          // static local x retains its previous value
33    c();          // global x also retains its value
34
```

**x** is different inside and outside the block.

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

```
35    cout << "local x in main is " << x << endl;
36
37    return 0;
38 }
39
40 void a( void )
41 {
42    int x = 25;  // initialized each time a is called
43
44    cout << endl << "local x in a is " << x
45         << " after entering a" << endl;
46    ++x;
47    cout << "local x in a is " << x
48         << " before exiting a" << endl;
49 }
50
51 void b( void )
52 {
53     static int x = 50; // Static initialization on
54                        // first time b is called.
55     cout << endl << "local static x is " << x
56          << " on entering b" << endl;
57     ++x;
58     cout << "local static x is " << x
59          << " on exiting b" << endl;
60 }
61
62 void c( void )
63 {
64    cout << endl << "global x is " << x
65         << " on entering c" << endl;
66    x *= 10;
67    cout << "global x is " << x << " on exiting c" << endl;
68 }
```

Local automatic variables are created and destroyed each time **a** is called.

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

Local static variables are not destroyed when the function ends.

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

Global variables are always accessible. Function **c** references the global **x.**

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

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

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

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

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

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

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

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

# References and Reference Parameters

- Call by value
  - Copy of data passed to function
  - Changes to copy do not change original
  - Used to prevent unwanted side effects
- Call by reference
  - Function can directly access data
  - Changes affect original
- Reference parameter alias for argument
  - **&** is used to signify a reference

```
void change( int &variable )
   { variable += 3; }
```

  - Adds 3 to the variable inputted

```
int y = &x.
```

  - A change to **y** will now affect **x** as well

```cpp
// Fig. 3.20: fig03_20.cpp
// Comparing call-by-value and call-by-reference
// with references.
#include <iostream>

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

int squareByValue( int );
void squareByReference( int & );

int main()
{
   int x = 2, z = 4;

   cout << "x = " << x << " before squareByValue\n"
        << "Value returned by squareByValue: "
        << squareByValue( x ) << endl
        << "x = " << x << " after squareByValue\n" << endl;

   cout << "z = " << z << " before squareByReference" << endl;
   squareByReference( z );
   cout << "z = " << z << " after squareByReference" << endl;

   return 0;
}

int squareByValue( int a )
{
   return a *= a;   // caller's argument not modified
}
```

```
32
33 void squareByReference( int &cRef )
34 {
35    cRef *= cRef;    // caller's argument modified
36 }
```

```
x = 2 before squareByValue
Value returned by squareByValue: 4
x = 2 after squareByValue

z = 4 before squareByReference
z = 16 after squareByReference
```

# Functions with Empty Parameter Lists

- Empty parameter lists
  - Either writing **`void`** or leaving a parameter list empty indicates that the function takes no arguments

```
void print();
```

or

```
void print( void );
```

  - Function **`print`** takes no arguments and returns no value

```
1  // Fig. 3.18: fig03_18.cpp
2  // Functions that take no arguments
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  void function1();
9  void function2( void );
10
11 int main()
12 {
13    function1();
14    function2();
15
16    return 0;
17 }
18
19 void function1()
20 {
21    cout << "function1 takes no arguments" << endl;
22 }
23
24 void function2( void )
25 {
26    cout << "function2 also takes no arguments" << endl;
27 }
```

```
function1 takes no arguments
function2 also takes no arguments
```

# 3.10 Default Arguments

- If function parameter omitted, gets default value
  - Can be constants, global variables, or function calls
  - If not enough parameters specified, rightmost go to their defaults
- Set defaults in function prototype

```
int defaultFunction( int x = 1,
   int y = 2, int z = 3 );
```

```cpp
// Fig. 3.23: fig03_23.cpp
// Using default arguments
#include <iostream>

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

int boxVolume( int length = 1, int width = 1, int height = 1 );

int main()
{
   cout << "The default box volume is: " << boxVolume()
        << "\n\nThe volume of a box with length 10,\n"
        << "width 1 and height 1 is: " << boxVolume( 10 )
        << "\n\nThe volume of a box with length 10,\n"
        << "width 5 and height 1 is: " << boxVolume( 10, 5 )
        << "\n\nThe volume of a box with length 10,\n"
        << "width 5 and height 2 is: " << boxVolume( 10, 5, 2 )
        << endl;

   return 0;
}

// Calculate the volume of a box
int boxVolume( int length, int width, int height )
{
   return length * width * height;
}
```

```
The default box volume is: 1

The volume of a box with length 10,
width 1 and height 1 is: 10

The volume of a box with length 10,
width 5 and height 1 is: 50

The volume of a box with length 10,
width 5 and height 2 is: 100
```

# Program Output

# 3.11 Function Overloading

- Function overloading
  - Having functions with same name and different parameters
  - Should perform similar tasks ( i.e., a function to square **int**s, and function to square **float**s).

```
int square( int x) {return x * x;}
float square(float x) { return x * x; }
```

  - Program chooses function by signature
    - signature determined by function name and parameter types
  - Can have the same return types

```
1  // Fig. 3.25: fig03_25.cpp
2  // Using overloaded functions
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  int square( int x ) { return x * x; }
9
10 double square( double y ) { return y * y; }
11
12 int main()
13 {
14    cout << "The square of integer 7 is " << square( 7 )
15         << "\nThe square of double 7.5 is " << square( 7.5 )
16         << endl;
17
18    return 0;
19 }
```

```
The square of integer 7 is 49
The square of double 7.5 is 56.25
```

End.