Cairo University
Faculty of Computers and Information
Computer Science Department

Programming-1 CS112
2017/2018

Functions

Dr. Amin Allam

1 Defining and calling functions

A *function* is a *block* of statements. A function must have a name, which follows the rule of naming identifiers. The statements inside any function are executed only when the function is *called* from another *calling* function. A function is called by a statement which consists of the function name followed by parentheses `()`. After the called function is executed, the calling function resumes its execution from the point it arrived. A function can be called any number of times, from the same or different calling functions, or not called at all. The *main()* function is a special function which is called by the operating system when the program starts.

```
1  void Fun1()              // The definition of function Fun1()
2  {
3      cout << "Fun1() starts." << endl;
4      cout << "Fun1() ends." << endl;
5  }
6  void Fun2()              // The definition of function Fun2()
7  {
8      cout << "Fun2() starts." << endl;
9      cout << "Fun2() ends." << endl;
10 }
11 void Fun3()              // The definition of function Fun3()
12 {
13     cout << "Fun3() starts." << endl;
14     cout << "Calling Fun1() from Fun3():" << endl;
15     Fun1();
16     cout << "Fun3() ends." << endl;
17 }
18 int main()               // The definition of function main()
19 {
20     cout << "main() starts." << endl;
21     cout << "Calling Fun1() from main():" << endl;
22     Fun1();
23     cout << "Calling Fun3() from main():" << endl;
24     Fun3();
25     cout << "main() ends." << endl;
26     return 0;
27 }
```

When this program is executed, its output will be as follows:

```
main() starts.
Calling Fun1() from main():
Fun1() starts.
Fun1() ends.
Calling Fun3() from main():
Fun3() starts.
Calling Fun1() from Fun3():
Fun1() starts.
Fun1() ends.
Fun3() ends.
main() ends;
```

The functions in the previous example do not have parameters, indicated by the empty parentheses. A function can have one or more parameters separated by commas. The default parameter passing method is passing by value which is explained in the following example:

```
1  void Fun(int y)          // Initially y=5 which is the value of x when calling Fun(x)
2  {
3     cout << y << endl;    // Prints 5 on screen when Fun(x) is called with x=5
4     y=7;                  // y value becomes 7, while the value of x does not change
5     cout << y << endl;    // Prints 7 on screen when Fun(x) is called
6     cout << x << endl;    ✖   // x is not visible from here
7  }
8
9  int main()
10 {
11    int x=5;
12    Fun(x);
13    cout << x << endl;    // Prints 5 on screen, because x value did not change
14    cout << y << endl;    ✖   // y is not visible from here
15    return 0;
16 }
```

In the previous example, we call `x` the *actual parameter*, while we call `y` the *formal parameter*. `x` and `y` are *local variables*. A local variable is visible only inside its block of statements.

A function can have a *return value* which can be received by the calling function. The type of the return value is specified before the function name in the *function definition*. A function returns its value by a *return* statement, which ends the function execution. If the function does not return a value, the return type is specified as *void* and no return statement exists, such as the previous example. A function can have at most one return value. The return value of the `main()` function is passed to the operating system and used for error reporting.

The following example illustrates the usage of a function with 3 integer parameters and an integer return type:

```
1  int MulMod(int a, int b, int c)
2  {
3     int r = (a * b) % c;
4     return r;
5  }
6  int main()
7  {
8     int x = 5, y = 7, z = 3;
9     int t = MulMod(x, y, z);       // set t = (5×7) mod 3 = 2
10    cout << t << endl;             // Print 2 on screen
11    int s = MulMod(y, 4, z-1);     // set s = (7×4) mod 2 = 0
12    cout << s << endl;             // Print 0 on screen
13    return 0;
14 }
```

2 Function declaration, default parameters, and overloading

In the previous examples, the *function definition* precedes function calls in the file. It is possible that function calls precedes the function definition such that the *function declaration (prototype)* precedes function calls. The parameter variable names can be omitted from function declarations.

```
1  int MulMul(int a, int b, int c) {return (a * b) * c;}
2  int MulMod(int, int, int);    // Variable names can be omitted from declaration
3  int MulAdd(int a, int b, int c) {return (a * b) + c;}
4
5  int main()
6  {
7     cout << MulMod(7, 4, 2) << endl; ✔ // Function declaration exists above
8     cout << MulMul(2, 3, 4) << endl; ✔ // Function definition exists above
9     cout << MulDiv(5, 8, 3) << endl; ✖ // No declaration or definition above
10    return 0;
11 }
12 int MulMod(int a, int b, int c) {return (a * b) % c;}
13 int MulDiv(int a, int b, int c) {return (a * b) / c;}
14 int MulAdd(int a, int b, int c) {return (a * b) + c;} ✖
15                                    // A function can not be defined twice
16 int MulMod(int a, int b, int c); ✔ // A function can be declared twice
```

It is possible to define default values for some formal parameters to be used in case actual parameters are not passed. Default values are specified in function definition only if function declaration does not exist, otherwise they are specified in function declaration only. If a default value for a formal parameter is specified, default values of subsequent formal parameters must be specified.

```
1  int MulMul(int a, int b, int c=2) {return (a * b) * c;} ✔
2                            // Default value can exist in definition if no  declaration
3  int MulMod(int, int=5, int=4);     ✔
4  int MulAdd(int a, int b=3, int c) {return (a * b) + c;} ✖
5                            // c must have a default value since b has one
6  int main()
7  {
8     cout << MulMod(7, 3, 2) << endl;   ✔   // Default values not used
9     cout << MulMod(7, 3)    << endl;   ✔   // MulMod(7,3,4)
10    cout << MulMod(7)       << endl;   ✔   // MulMod(7,5,4)
11    cout << MulMod()        << endl;   ✖   // a does not have a default value
12    cout << MulMul(7, 3, 5) << endl;   ✔   // Default value not used
13    cout << MulMul(7, 3)    << endl;   ✔   // MulMul(7,3,2)
14    cout << MulMul(7)       << endl;   ✖   // b does not have a default value
15    return 0;
16 }
17 int MulMod(int a, int b=5, int c=4) {return (a * b) % c;} ✖
18                                       // Default values in declaration only
```

It is possible to *overload* functions. Overloading functions is defining functions with the same name but they differ in the number or the types of parameters. It is not possible to define functions with the same name that differ only in the return type.

```
1  int    Mul(int a, int b, int c) {return a * b * c;}
2  int    Mul(int a, int b)        {return a * b;}  ✔
3  double Mul(int a, int b)        {return a * b;}  ✖
4                            // Overloaded functions can not differ only in return type
5  double Mul(int a, double b)     {return a * b;}  ✔
6
7  int main()
8  {
9     cout << Mul(7, 3, 2)   << endl;   ✔
10    cout << Mul(7, 3, 2.7) << endl;   ✔   // Best match is: Mul(int, int, int)
11    cout << Mul(7, 3)      << endl;   ✔   // Best match is: Mul(int, int)
12    cout << Mul(7, 3.0)    << endl;   ✔   // Best match is: Mul(int, double)
13    cout << Mul(7.0, 3.0)  << endl;   ✔   // Best match is: Mul(int, double)
14    return 0;
15 }
```

3 Scope and lifetime of variables

A variable *scope* defines where it is visible. A variable *lifetime* defines when it is constructed and destroyed.
All variables defined inside a function are *local* variables to this function. A local variable is constructed each time the function is called, and destroyed each time the function ends. It is visible only after its declaration and inside the block of statements {} to which it belongs. If the *static* modifier precedes a local variable declaration, it will be constructed and initialized once, and destroyed only at the end of the program. Static variables are initialized to zero be default.
All variables defined outside all functions are *global* variables. Global variables are constructed at the beginning of the program and destroyed at the end of the program. They are visible inside all functions after its declaration. Global variables are initialized to zero be default. If the *static* modifier precedes a global variable, it will be visible only inside its file.
A variable may become invisible in a block of statements because a variable with the same name exists in that block.

```
1  int a, b=5;                 // Global variables, a initialized to 0
2  static int c;               // c is not visible to other files
3
4  void Fun(int x)
5  {
6     int y, z=7;                    // Local variables, y not initialized
7     static int r, t=3;             // Static local variables, r initialized to 0
8                                    //   r and t are initialized only once per program
9
10    a++; b++; c++; x++; y++; z++; r++; t++;
11
12    if(t==4) {int z; z=20;}    // The above z is not affected
13
14    cout <<  "x=" << x << " y=" << y << " z=" << z
15         << " r=" << r << " t=" << t << endl;
16 }
17
18 int main()
19 {
20    cout << a << " " << b << " " << c << endl;   ✔   // Prints 0 5 0
21    Fun(2);                          // Prints x=3 y=(garbage value) z=8 r=1 t=4
22    cout << a << " " << b << " " << c << endl;   ✔   // Prints 1 6 1
23    Fun(6);                          // Prints x=7 y=(garbage value) z=8 r=2 t=5
24    cout << a << " " << b << " " << c << endl;   ✔   // Prints 2 7 2
25    cout << x << " " << y << " " << z << endl;   ✖   // Not visible here
26    cout << r << " " << t << endl;               ✖   // Not visible here
27    return 0;
28 }
```

4 Recursion

A function can recursively call itself with different parameter values. Each function call has its separate local variables. A recursive function must contain a non-recursive base case.

```
1 int Factorial(int n)    // Returns n×(n−1)×(n−2)×···×1 = n×Factorial(n−1)
2 {
3    if(n<=1) return 1;          // Factorial(0)=1 and Factorial(1)=1
4    int f=Factorial(n-1);       // Compute Factorial(n−1)
5    int r=n*f;                  // Multiply it by n
6    return r;
7 }
```

For example, calling `Factorial(1)` will return `1`.
Calling `Factorial(2)` will call `Factorial(1)`, which returns `1`, and then multiplies it by `2` and returns `2`.
Calling `Factorial(3)` will call `Factorial(2)`, which returns `2` as we see from the previous discussion, then multiplies it by `3` and returns `6`.
Note that each function call has a separate list of the local variables `n`, `f`, `r` which are different from the local variables of the other function calls, although they have the same name.

The following is another example on recursion:

```
1 double Power(double x, int p)          // Returns x^p where p is integer
2 {
3    if(p==0) return 1;                  // Base case: x^0 = 1
4    if(p<0) return 1.0/Power(x,-p);     // x^-p = 1/x^p
5    return x*Power(x,p-1);              // x^p = x × x^(p−1)
6 }
```