CS118 – Programming Fundamentals

Lecture # 20
Tuesday, October 30, 2019
FALL 2019
FAST – NUCES, Faisalabad Campus

Zain Iqbal

Local and Global Variables

- **Local variable:** Defined within a function or block; accessible only within the function or block
- Other functions and blocks can define variables with the **same name**
- When a function is called, local variables in the calling function are **not accessible** from within the called function
- C++ **does not** allow the **nesting of functions**. That is, you cannot include the definition of one function in the body of another function.

Local and Global Variables

- **Global variable:** A variable defined outside all functions; it is accessible to all functions within its scope
- Easy way to **share large amounts of data** between functions
- **Scope of a global variable** is from its point of definition to the program end
  - Use cautiously

Local Variable Lifetime

- A local variable only **exists** while its defining function is executing
- Local variables are **destroyed** when the function terminates
- Data **cannot be retained** in local variables defined in a function between calls to the function

Initializing Local and Global Variables

- **Local** variables **must be initialized** by the programmer
- **Global** variables are initialized to **0 (numeric)** or NULL (character) when the variable is defined

Local and Global Variable Names

- Local variables **can** have **same names** as global variables
- When a function contains a local variable that has the same name as a global variable, the global variable is unavailable from within the function
- The local definition "hides" or "shadows" the global definition

If Local and Global Variable have different name


```cpp
#include <iostream>
using namespace std;
int t;
void funOne(int& a);
int main()
{
int x = 15; //Line 1
cout << "Line 2: In main: t = " << t << endl; //Line 2
funOne(x); //Line 3
cout << "Line 4: In main after funOne: "
<< " t = " << t << endl; //Line 4
return 0; //Line 5
}
void funOne(int& a)
{
cout << "Line 6: In funOne: a = " << a
<< " and t = " << t << endl; //Line 6
a = a + 12; //Line 7
cout << "Line 8: In funOne: a = " << a
<< " and t = " << t << endl; //Line 8
t = t + 13; //Line 9
cout << "Line 10: In funOne: a = " << a
<< " and t = " << t << endl; //Line 10
}
```

```
Line 2: In main: t = 0
Line 6: In funOne: a = 15 and t = 0
Line 8: In funOne: a = 27 and t = 0
Line 10: In funOne: a = 27 and t = 13
Line 4: In main after funOne:  t = 13
```

If Local and Global have same name


```
#include <iostream>
using namespace std;
int t;
void funOne(int& a);
int main()
{
t = 15; //Line 1
cout << "Line 2: In main: t = " << t << endl; //Line 2
funOne(t); //Line 3
cout << "Line 4: In main after funOne: "
<< " t = " << t << endl; //Line 4
return 0; //Line 5
}
void funOne(int& a)
{
cout << "Line 6: In funOne: a = " << a
<< " and t = " << t << endl; //Line 6
a = a + 12; //Line 7
cout << "Line 8: In funOne: a = " << a
<< " and t = " << t << endl; //Line 8
t = t + 13; //Line 9
cout << "Line 10: In funOne: a = " << a
<< " and t = " << t << endl; //Line 10
}
```

```
Line 2: In main: t = 15
Line 6: In funOne: a = 15 and t = 15
Line 8: In funOne: a = 27 and t = 27
Line 10: In funOne: a = 40 and t = 40
Line 4: In main after funOne:  t = 40
```

Static Local Variables

- **Local variables**
  - Only exist while the function is executing
  - Are redefined each time function is called
  - Lose their contents when function terminates
- **static local variables**
  - Are defined with key word static

    static int counter;
  - Are defined and initialized **only the first time** the function is executed
  - Retain their contents between function calls
  - Better to initialize when declared

    static int counter = 0 ;

static **variable illustrated**


```
//Program: Static and automatic variables

#include <iostream>

using namespace std;

void test();

int main()
{
    int count;

    for (count = 1; count <= 5; count++)
        test();

    return 0;
}

void test()
{
    static int x = 0;
    int y = 10;

    x = x + 2;
    y = y + 1;

    cout << "Inside test x = " << x << " and y = "
         << y << endl;
}
```

Sample Run:

```
Inside test x = 2 and y = 11
Inside test x = 4 and y = 11
Inside test x = 6 and y = 11
Inside test x = 8 and y = 11
Inside test x = 10 and y = 11
```

Default Arguments

- Values passed automatically if arguments are missing from the function call
- Must be a constant declared in prototype

  void evenOrOdd(int = 0);
- Multi-parameter functions may have default arguments for some or all of them

  int getSum(int, int=0, int=0);
- If you specify a value to default parameter then it will be used otherwise default value will be used

Default Arguments

- If not all parameters to a function have default values, the ones without defaults must be declared first in the parameter list

```
int getSum(int, int=0, int=0);  // OK
int getSum(int, int=0, int); // wrong!
```

- When an argument is omitted from a function call, all arguments after it must also be omitted

```
sum = getSum(num1, num2);    // OK
sum = getSum(num1, , num3);  // wrong!
```

- **Constant** value **can't be assigned** to **reference parameter**

```
void func(int x, int& y=16, double z=34);
```

```
#include <iostream>
#include <iomanip>

using namespace std;

int volume(int l = 1, int w = 1, int h = 1);
void funcOne(int& x, double y = 12.34, char z = 'B');

int main()
{
    int a = 23;
    double b = 48.78;
    char ch = 'M';

    cout << fixed << showpoint;
    cout << setprecision(2);

    cout << "Line 1: a = " << a << ", b = "
         << b  << ", ch = " << ch << endl;          //Line 1
    cout << "Line 2: Volume = " << volume()
         << endl;                                   //Line 2
    cout << "Line 3: Volume = " << volume(5, 4)
         << endl;                                   //Line 3
    cout << "Line 4: Volume = " << volume(34)
         << endl;                                   //Line 4
    cout << "Line 5: Volume = "
         << volume(6, 4, 5) << endl;                //Line 5

    funcOne(a);                                     //Line 6
    funcOne(a, 42.68);                              //Line 7
    funcOne(a, 34.65, 'Q');                         //Line 8

    cout << "Line 9: a = " << a << ", b = "
         << b << ", ch = " << ch << endl;           //Line 9

    return 0;
}
```

```
int volume(int l, int w, int h)
{
    return l * w * h;                                    //Line 10
}

void funcOne(int& x, double y, char z)
{
    x = 2 * x;                                           //Line 11
    cout << "Line 12: x = " << x << ", y = "
         << y << ", z = " << z << endl;                  //Line 12
}
```

Sample Run:

```
Line 1: a = 23, b = 48.78, ch = M
Line 2: Volume = 1
Line 3: Volume = 20
Line 4: Volume = 34
Line 5: Volume = 120
Line 12: x = 46, y = 12.34, z = B
Line 12: x = 92, y = 42.68, z = B
Line 12: x = 184, y = 34.65, z = Q
Line 9: a = 184, b = 48.78, ch = M
```

Overloading Functions

- **Overloaded functions** are two or more functions that have the **same name**, but **different parameter lists**
- Can be used to create functions that perform the same task, but take **different parameter types** or **different number of parameters**
- **Compiler will determine** which version of function to call by argument and parameter list
- Important thing is Formal parameter should be different either their datatype, number or position

Overloaded Functions - Examples

- If a program has these overloaded functions:

```
int getDimensions(int)              // 1
int getDimensions(int, int)         // 2
int getDimensions(int, double)      // 3
int getDimensions(double, double)   // 4
```

- The compiler will use them as follows:

```
int length, width;
double base, height;
getDimensions(length);              // 1
getDimensions(length, width);       // 2
getDimensions(length, height);      // 3
getDimensions(height, base);        // 4
```

Overloaded Functions - Examples


```
void functionXYZ()
void functionXYZ(int x, double y)
void functionXYZ(double one, int y)
void functionXYZ(int x, double y, char ch)
```

- Consider the following function headings to overload the function functionABC:

```
void functionABC(int x, double y)
int functionABC(int x, double y)
```

The exit() Function

- Terminates execution of a program
- Can be called from any function
- Can pass a value to operating system to indicate status of program execution
- Usually used for abnormal termination of program
- Requires **cstdlib** header file

What will be the output of the following Program's ?

```
void find(int a, int& b, int& c,)

int main()
{
    int one, two, three;

    one = 5;
    two = 10;
    three = 15;

    find(one, two, three);
    cout << one << ", " << two << ", " << three << endl;

    find(two, one, three);
    cout << one << ", " << two << ", " << three << endl;

    find(three, two, one);
    cout << one << ", " << two << ", " << three << endl;

    find(two, three, one);
    cout << one << ", " << two << ", " << three << endl;

    return 0;
 }

void find(int a, int& b, int& c)
{
    int temp;

    c = a + b;
    temp = a;
    a = b;
    b = 2 * temp;
}
```

```
5, 10, 15
20, 10, 15
25, 30, 15
45, 30, 60
```

```
int x;

void summer(int&, int);
void fall(int, int&);

int main()
{
    int intNum1 = 2;
    int intNum2 = 5;
    x = 6;

    summer(intNum1, intNum2);
    cout << intNum1 << " " << intNum2 << " " << x << endl;

    fall(intNum1, intNum2);
    cout << intNum1 << " " << intNum2 << " " << x << endl;
    return 0;
}

void summer(int& a, int b)
{
    int intNum1;
    intNum1 = b + 12;
    a = 2 * b + 5;
    b = intNum1 + 4;
}

void fall(int u, int& v)
{
    int intNum2;
    intNum2= x;
    v = intNum2 * 4;
    x = u - v;
}
```

```
15 5 6
15 24 -9
```

Summary

- Functions (modules) are miniature programs
  - Divide a program into manageable tasks
- C++ provides the standard functions
- Two types of user-defined functions: value-returning functions and void functions
- Variables defined in a function heading are called formal parameters
- Expressions, variables, or constant values in a function call are called actual parameters

Summary (cont'd.)

- In a function call, the number of actual parameters and their types must match with the formal parameters in the order given
- To call a function, use its name together with the actual parameter list
- Function heading and the body of the function are called the definition of the function
- A value-returning function returns its value via the **return** statement

Summary (cont'd.)

- A prototype is the function heading without the body of the function; prototypes end with the semicolon
- Prototypes are placed before every function definition, including **main**
- User-defined functions execute only when they are called
- In a call statement, specify only the actual parameters, not their data types

Questions