COMSATS University
Islamabad
(Lahore Campus)

CSC103- Programming Fundamentals

MS. MAHWISH WAQAS

MAWISH.WAQAS@CUILAHORE.EDU.PK

Chapter 6:
User-Defined Functions

Scope of an Identifier

- Scope of an identifier: where in the program the identifier is accessible
- Local identifier: identifiers declared within a function (or block)
- Global identifier: identifiers declared outside of every function definition
- C++ does not allow nested functions
  - Definition of one function cannot be included in the body of another function

Scope of an Identifier (cont'd.)

- Rules when an identifier is accessed:
  - Global identifiers are accessible by a function or block if:
    - Declared before function definition
    - Function name different from identifier
    - Parameters to the function have different names
    - All local identifiers have different names

Scope of an Identifier (cont'd.)

- Rules when an identifier is accessed (cont'd.):
  - Nested block
    - Identifier accessible from declaration to end of block in which it is declared
    - Within nested blocks if no identifier with same name exists
  - Scope of function name similar to scope of identifier declared outside any block
    - i.e., function name scope = global variable scope

```
#include <iostream>
using namespace std;
const double RATE = 10.50;
int z;
double t;
void one(int x, char y);
void two(int a, int b, char x);
void three(int one, double y, int z);
int main()
{
int num, first;
double x, y, z;
char name, last;
.
.
return 0;
}
void one(int x, char y)
{
.
.
}
int w;
void two(int a, int b, char x)
{
int count;
.
.
}
void three(int one, double y, int z)
{
char ch;
int a;
.
.
//Block four
{
int x;
char a;
.
.
}//end Block four
.
.
}
```

Identifier	Visibility in one	Visibility in two	Visibility in three	Visibility in Block four	Visibility in main
RATE (before main)	Y	Y	Y	Y	Y
z (before main)	Y	Y	N	N	N
t (before main)	Y	Y	Y	Y	Y
main	Y	Y	Y	Y	Y
local variables of main	N	N	N	N	Y
one (function name)	Y	Y	N	N	Y
x (one's formal parameter)	Y	N	N	N	N
y (one's formal parameter)	Y	N	N	N	N
w (before function two)	N	Y	Y	Y	N
two (function name)	Y	Y	Y	Y	Y
a (two's formal parameter)	N	Y	N	N	N
b (two's formal parameter)	N	Y	N	N	N
x (two's formal parameter)	N	Y	N	N	N
local variables of two	N	Y	N	N	N
three (function name)	Y	Y	Y	Y	Y
one (three's formal parameter)	N	N	Y	Y	N
y (three's formal parameter)	N	N	Y	Y	N
z (three's formal parameter)	N	N	Y	Y	N
ch (three's local variable)	N	N	Y	Y	N
a (three's local variable)	N	N	Y	N	N
x (Block four's local variable)	N	N	N	Y	N
a (Block four's local variable)	N	N	N	Y	N

Scope of an Identifier (cont'd.)

- Some compilers initialize global variables to default values
- Scope resolution operator in C++ is `::`
- By using the scope resolution operator
  - A global variable declared before the definition of a function (or block) can be accessed by the function (or block)
  - Even if the function (or block) has an identifier with the same name as the global variable

Scope of an Identifier (cont'd.)

- To access a global variable declared after the definition of a function, the function must not contain any identifier with the same name
  - Reserved word `extern` indicates that a global variable has been declared elsewhere

Global Variables, Named Constants, and Side Effects

- Using global variables causes side effects
- A function that uses global variables is not independent
- If more than one function uses the same global variable:
  - Can be difficult to debug problems with it
  - Problems caused in one area of the program may appear to be from another area
- Global named constants have no side effects

```
//Global variable

#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
}
```

Sample Run:

```
Line 8: In main: t = 15
Line 15: In funOne: a = 15 and t = 15
Line 17: In funOne: a = 27 and t = 27
Line 19: In funOne: a = 40 and t = 40
Line 10: In main after funOne: t = 40
```

Static and Automatic Variables

- Automatic variable: memory is allocated at block entry and deallocated at block exit
  - By default, variables declared within a block are automatic variables
- Static variable: memory remains allocated as long as the program executes
  - Global variables declared outside of any block are static variables

Static and Automatic Variables (cont'd.)

- Can declare a static variable within a block by using the reserved word static
- Syntax:

```
static dataType identifier;
```

- Static variables declared within a block are local to the block
  - Have same scope as any other local identifier in that block

```
//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
```

Debugging:
Using Drivers and Stubs

- Driver program: separate program used to test a function
- When results calculated by one function are needed in another function, use a function stub
- Function stub: a function that is not fully coded

Function Overloading: An Introduction

- In a C++ program, several functions can have the same name
- Function overloading: creating several functions with the same name
- Function signature: the name and formal parameter list of the function
  - Does *not* include the return type of the function

Function Overloading (cont'd.)

- Two functions are said to have different formal parameter lists if both functions have either:
  - A different number of formal parameters
  - If the number of formal parameters is the same, but the data type of the formal parameters differs in at least one position
- Overloaded functions must have different function signatures

Function Overloading (cont'd.)

- The parameter list supplied in a call to an overloaded function determines which function is executed

```
int larger(int x, int y);
char larger(char first, char second);
double larger(double u, double v);
string larger(string first, string second);
```

- Function overloading is used when you have the same action for different sets of data. Of course, for function overloading to work, you must give the definition of each function.

Functions with Default Parameters

- In a function call, the number of actual and formal parameters must be the same
  - C++ relaxes this condition for functions with default parameters
- Can specify the value of a default parameter in the function prototype
- If you do not specify the value for a default parameter when calling the function, the default value is used

Functions with Default Parameters (cont'd.)

- All default parameters must be the rightmost parameters of the function
- If a default parameter value is not specified:
  - You must omit all of the arguments to its right
- Default values can be constants, global variables, or function calls
- Cannot assign a constant value as a default value to a reference parameter

Functions with Default Parameters

Consider the following function prototype:

```
void funcExp(int x, int y, double t, char z = 'A', int u = 67, char v = 'G',
double w = 78.34);
```

The function funcExp has seven parameters. The parameters z, u, v, and w are default parameters. If no values are specified for z, u, v, and w in a call to the function funcExp, their default values are used.

Suppose you have the following statements:

```
int a, b;
char ch;
double d;
```

The following function calls are legal:

1. funcExp(a, b, d);
2. funcExp(a, 15, 34.6, 'B', 87, ch);
3. funcExp(b, a, 14.56, 'D');

Functions with Default Parameters

Consider the following function prototype:

```
void funcExp(int x, int y, double t, char z = 'A', int u = 67, char v = 'G',
double w = 78.34);
```

The following function calls are illegal:

1. funcExp(a, 15, 34.6, 46.7);

2. funcExp(b, 25, 48.76, 'D', 4567, 78.34);

In statement 1, because the value of z is omitted, all other default values must be omitted.

In statement 2, because the value of v is omitted, the value of w should be omitted, too.

Functions with Default Parameters

The following are illegal function prototypes with default parameters:

1. void funcOne(int x, double z = 23.45, char ch, int u = 45);
2. int funcTwo(int length = 1, int width, int height = 1);
3. void funcThree(int x, int& y = 16, double z = 34);

In statement 1, because the second parameter z is a default parameter, all other parameters after z must be default parameters.
In statement 2, because the first parameter is a default parameter, all parameters must be the default parameters.
In statement 3, a constant value cannot be assigned to y because y is a reference parameter.

Summary (cont'd.)

- Void functions do not have a data type
  - Void functions are always called *standalone*
- Two types of formal parameters:
  - A value parameter receives a copy of its corresponding actual parameter
  - A reference parameter receives the memory address of its corresponding actual parameter
- Variables declared within a function (or block) are called local variables

Summary (cont'd.)

- Variables declared outside of every function definition (and block) are global variables
- Automatic variable: variable for which memory is allocated on function/block entry and deallocated on function/block exit
- Static variable: memory remains allocated throughout the execution of the program
- C++ functions can have default parameters