C++ Programming

Objectives

In this chapter, you will:

- Learn how to construct and use void functions in a program
- Discover the difference between value and reference parameters
- Explore reference parameters and value-returning functions
- Learn about the scope of an identifier

Objectives (continued)

- Examine the difference between local and global identifiers
- Discover static variables
- Learn function overloading
- Explore functions with default parameters

Void Functions

- Void functions and value-returning functions have similar structures
  - Both have a heading part and a statement part
- User-defined void functions can be placed either before or after the function `main`
- If user-defined void functions are placed after the function `main`
  - The function prototype must be placed before the function `main`

Void Functions (continued)

- A void function does not have a return type
  - `return` statement without any value is typically used to exit the function early
- Formal parameters are optional
- A call to a void function is a stand-alone statement

Void Functions without Parameters

- Function definition syntax:

```
void functionName()
{
     statements
}
```

- `void` is a reserved word
- Function call syntax:

```
functionName();
```

Void Functions with Parameters

- Function definition syntax:

```
void functionName(formal parameter list)
{
    statements
}
```

- Formal parameter list syntax:

```
dataType& variable, dataType& variable, ...
```

- Function call syntax:

```
functionName(actual parameter list);
```

- Actual parameter list syntax:

```
expression or variable,expression or variable, ...
```

Void Functions with Parameters (continued)

EXAMPLE 7-2

```
void funexp(int a, double b, char c, int x)
{
    .
    .
    .
}
```

The function `funexp` has four parameters.

EXAMPLE 7-3

```
void expfun(int one, int& two, char three, double& four)
{
     .
     .
     .
}
```

The function `expfun` has four parameters: (1) `one`, a value parameter of type `int`; (2) `two`, a reference parameter of type `int`; (3) `three`, a value parameter of type `char`, and (4) `four`, a reference parameter of type `double`.

Void Functions with Parameters (continued)

- Value parameter: a formal parameter that receives a copy of the content of corresponding actual parameter
- Reference parameter: a formal parameter that receives the location (memory address) of the corresponding actual parameter

Value Parameters

- If a formal parameter is a value parameter
  - The value of the corresponding actual parameter is copied into it
- The value parameter has its own copy of the data
- During program execution
  - The value parameter manipulates the data stored in its own memory space

Reference Variables as Parameters

- If a formal parameter is a reference parameter
  - It receives the memory address of the corresponding actual parameter
- A reference parameter stores the address of the corresponding actual parameter
- During program execution to manipulate data
  - The address stored in the reference parameter directs it to the memory space of the corresponding actual parameter

Reference Variables as Parameters (continued)

- Reference parameters can:
  - Pass one or more values from a function
  - Change the value of the actual parameter
- Reference parameters are useful in three situations:
  - Returning more than one value
  - Changing the actual parameter
  - When passing the address would save memory space and time

Calculate Grade

```
//This program reads a course score and prints the
//associated course grade.

#include <iostream>
using namespace std;

void getScore(int& score);
void printGrade(int score);

int main()
{
    int courseScore;

    cout << "Line 1: Based on the course score, \n"
         << "    this program computes the "
         << "course grade." << endl;                    //Line 1

    getScore(courseScore);                              //Line 2

    printGrade(courseScore);                            //Line 3

    return 0;
}
```

Calculate Grade (continued)

```
void getScore(int& score)
{
    cout << "Line 4: Enter course score: ";          //Line 4
    cin >> score;                                    //Line 5
    cout << endl << "Line 6: Course score is "
         << score << endl;                           //Line 6
}

void printGrade(int cScore)
{
    cout << "Line 7: Your grade for the course is "; //Line 7

    if (cScore >= 90)                                //Line 8
        cout << "A." << endl;
    else if (cScore >= 80)
        cout << "B." << endl;
    else if(cScore >= 70)
        cout << "C." << endl;
    else if (cScore >= 60)
        cout << "D." << endl;
    else
        cout << "F." << endl;
}
```

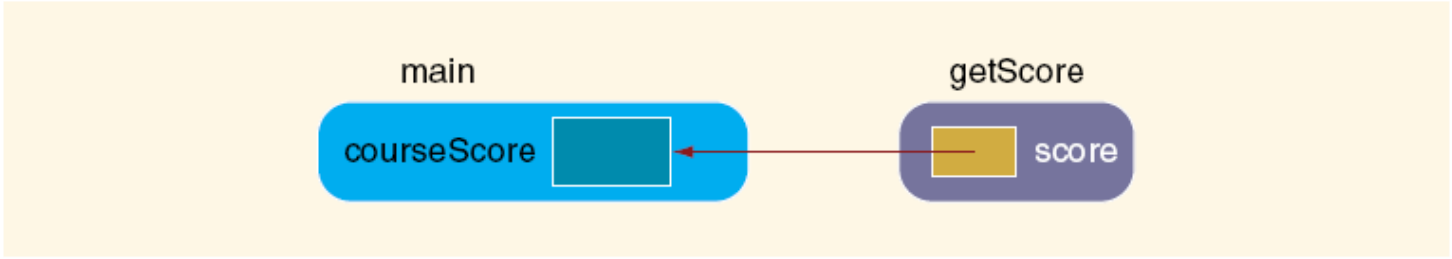

FIGURE 7-1 Variable `courseScore` and the parameter `score`

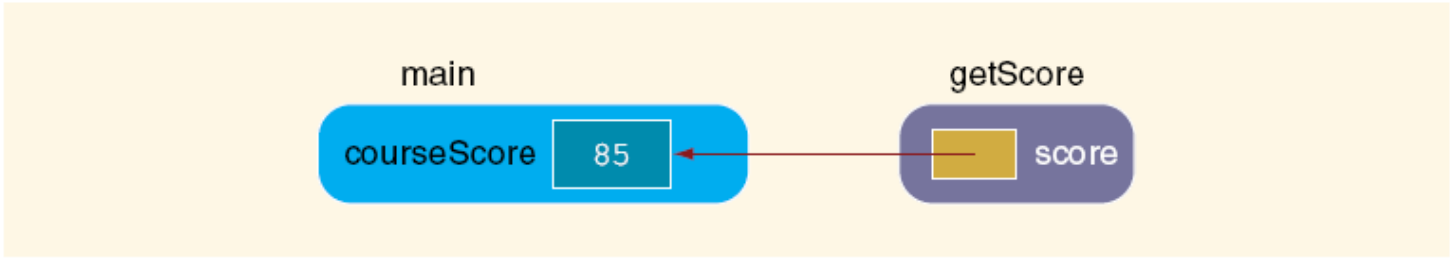

FIGURE 7-2 Variable `courseScore` and the parameter `score` after the statement in Line 5 executes

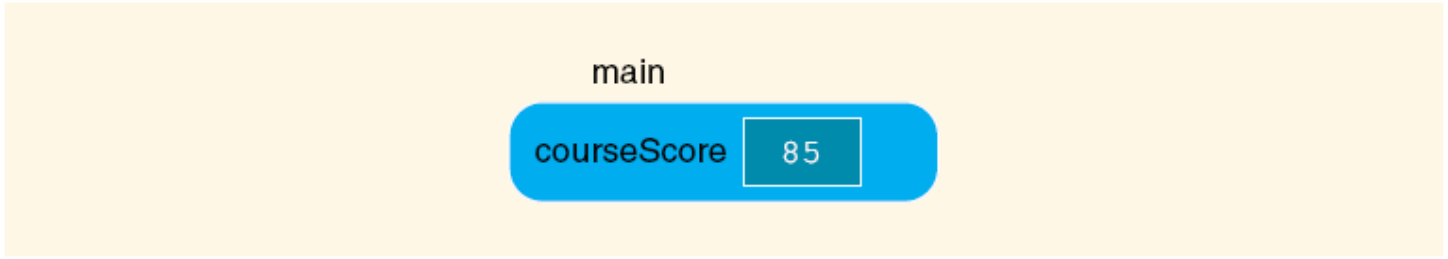

FIGURE 7-3 Variable `courseScore` after the statement in Line 6 is executed and control goes back to `main`

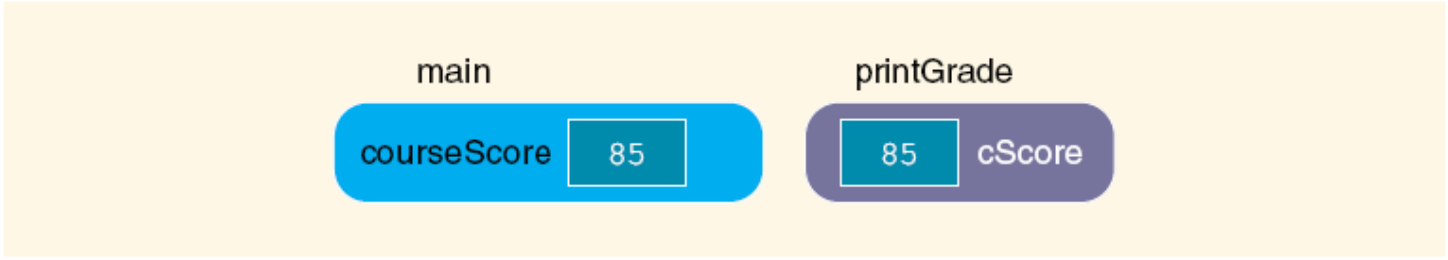

FIGURE 7-4 Variable `courseScore` and the parameter `cScore`

Value and Reference Parameters and Memory Allocation

- When a function is called
  - Memory for its formal parameters and variables declared in the body of the function (called local variables) is allocated in the function data area
- In the case of a value parameter
  - The value of the actual parameter is copied into the memory cell of its corresponding formal parameter

Value and Reference Parameters and Memory Allocation (continued)

- In the case of a reference parameter
  - The address of the actual parameter passes to the formal parameter
- Content of formal parameter is an address
- During execution, changes made by the formal parameter permanently change the value of the actual parameter
- Stream variables (e.g., `ifstream`) should be passed by reference to a function

EXAMPLE 7-7

```
#include <iostream>

using namespace std;

void funOne(int a, int& b, char v);
void funTwo(int& x, int y, char& w);

int main()
{
    int num1, num2;
    char ch;

    num1 = 10;                                          //Line 1
    num2 = 15;                                          //Line 2
    ch = 'A';                                           //Line 3

    cout << "Line 4: Inside main: num1 = " << num1
         << ", num2 = " << num2 << ", and ch = "
         << ch << endl;                                 //Line 4

    funOne(num1, num2, ch);                             //Line 5

    cout << "Line 6: After funOne: num1 = " << num1
         << ", num2 = " << num2 << ", and ch = "
         << ch << endl;                                 //Line 6

    funTwo(num2, 25, ch);                               //Line 7

    cout << "Line 8: After funTwo: num1 = " << num1
         << ", num2 = " << num2 << ", and ch = "
         << ch << endl;                                 //Line 8

    return 0;
}
```

```
void funOne(int a, int& b, char v)
{
    int one;

    one = a;                                                    //Line 9
    a++;                                                        //Line 10
    b = b * 2;                                                  //Line 11
    v = 'B';                                                    //Line 12

    cout << "Line 13: Inside funOne: a = " << a
         << ", b = " << b << ", v = " << v
         << ", and one = " << one << endl;                      //Line 13
}

void funTwo(int& x, int y, char& w)
{
    x++;                                                        //Line 14
    y = y * 2;                                                  //Line 15
    w = 'G';                                                    //Line 16

    cout << "Line 17: Inside funTwo: x = " << x
         << ", y = " << y << ", and w = " << w
         << endl;                                               //Line 17
}
```

Sample Run:

```
Line 4: Inside main: num1 = 10, num2 = 15, and ch = A
Line 13: Inside funOne: a = 11, b = 30, v = B, and one = 10
Line 6: After funOne: num1 = 10, num2 = 30, and ch = A
Line 17: Inside funTwo: x = 31, y = 50, and w = G
Line 8: After funTwo: num1 = 10, num2 = 31, and ch = G
```

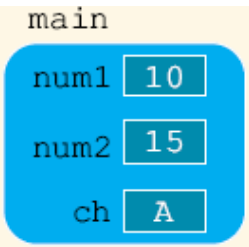

FIGURE 7-5 Values of the variables after the statement in Line 3 executes

```
one = a;                                                    //Line 9
```

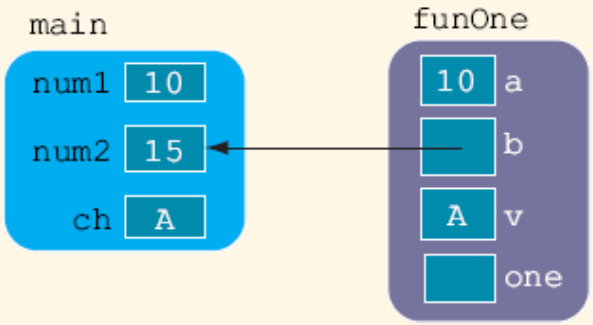

FIGURE 7-6 Values of the variables just before the statement in Line 9 executes

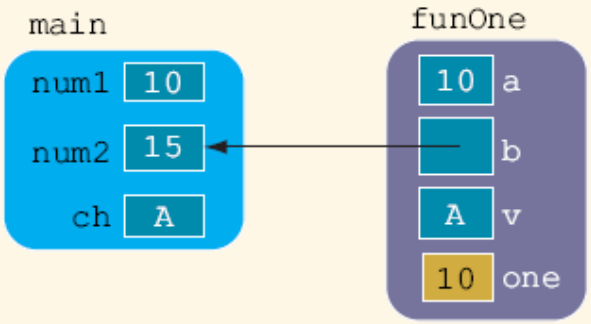

FIGURE 7-7 Values of the variables after the statement in Line 9 executes

```
a++;                                                    //Line 10
b = b * 2;                                              //Line 11
v = 'B';                                                //Line 12
```

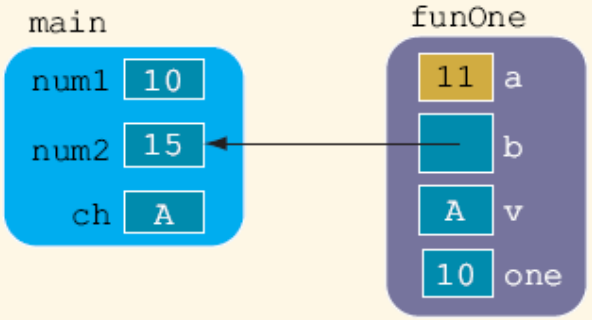

FIGURE 7-8 Values of the variables after the statement in Line 10 executes

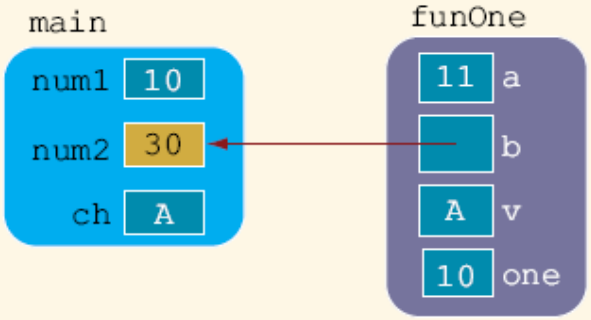

FIGURE 7-9 Values of the variables after the statement in Line 11 executes

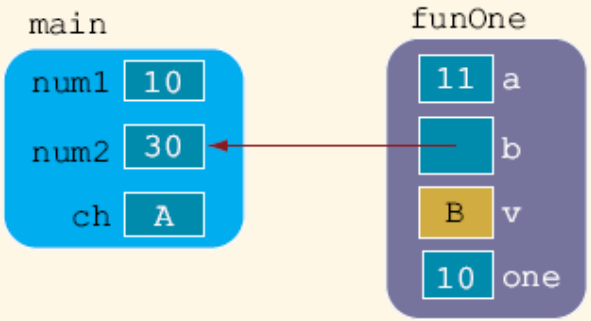

FIGURE 7-10 Values of the variables after the statement in Line 12 executes

```
cout << "Line 13: Inside funOne: a = " << a
     << ", b = " << b << ", v = " << v
     << ", and one = " << one << endl;          //Line 13
```

The statement in Line 13 produces the following output:

```
Line 13: Inside funOne: a = 11, b = 30, v = B, and one = 10
```

```
cout << "Line 6: After funOne: num1 = " << num1
     << ", num2 = " << num2 << ", and ch = "
     << ch << endl;                             //Line 6
```

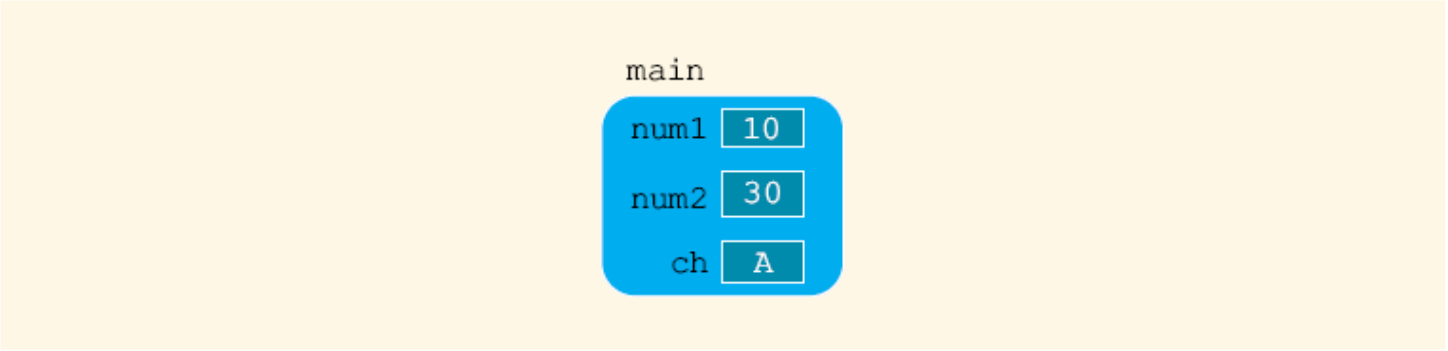

FIGURE 7-11 Values of the variables when control goes back to Line 6

Line 6 produces the following output:

```
Line 6: After funOne: num1 = 10, num2 = 30, and ch = A
```

```
x++;                                                    //Line 14
y = y * 2;                                              //Line 15
```

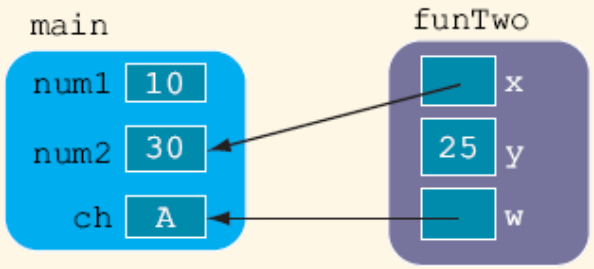

FIGURE 7-12 Values of the variables before the statement in Line 14 executes

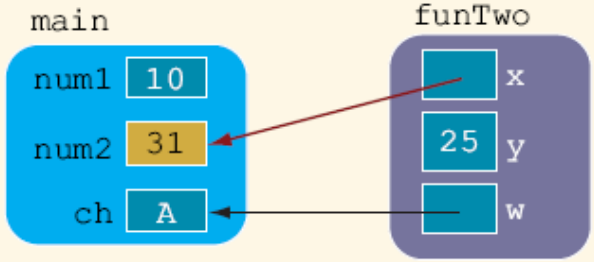

FIGURE 7-13 Values of the variables after the statement in Line 14 executes

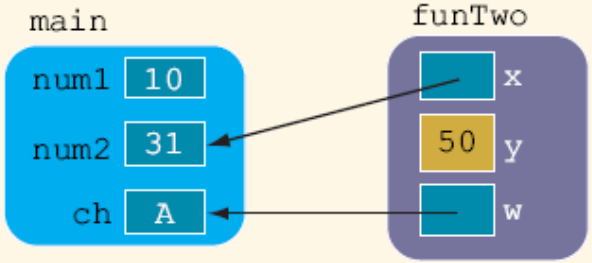

FIGURE 7-14 Values of the variables after the statement in Line 15 executes

```
w = 'G';                                                    //Line 16

cout << "Line 17: Inside funTwo: x = " << x
     << ", y = " << y << ", and w = " << w
     << endl;                                               //Line 17
```

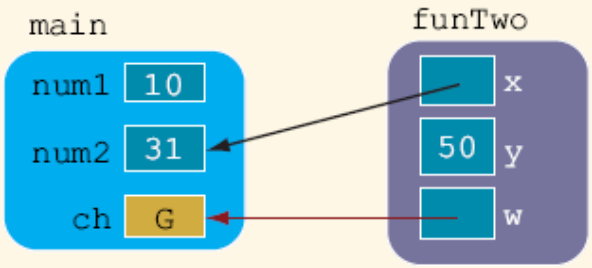

FIGURE 7-15 Values of the variables after the statement in Line 16 executes

Line 17 produces the following output:

```
Line 17: Inside funTwo: x = 31, y = 50, and w = G
```

```
cout << "Line 8: After funTwo: num1 = " << num1
     << ", num2 = " << num2 << ", and ch = "
     << ch << endl;                                         //Line 8
```

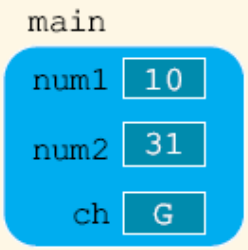

FIGURE 7-16 Values of the variables when control goes to Line 8

```
Line 8: After funTwo: num1 = 10, num2 = 31, and ch = G
```

Reference Parameters and Value-Returning Functions

- You can also use reference parameters in a value-returning function
  - Not recommended
- By definition, a value-returning function returns a single value
  - This value is returned via the return statement
- If a function needs to return more than one value, you should change it to a void function and use the appropriate reference parameters to return the values

Scope of an Identifier

- The scope of an identifier refers to where in the program an 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
  - The definition of one function cannot be included in the body of another function

In general, the following rules apply when an identifier is accessed:

1. Global identifiers (such as variables) are accessible by a function or a block if:
   a. The identifier is declared before the function definition (block),
   b. The function name is different from the identifier,
   c. All parameters of the function have names different than the name of the identifier, and
   d. All local identifiers (such as local variables) have names different than the name of the identifier.
2. **(Nested Block)** An identifier declared within a block is accessible:
   a. Only within the block from the point at which it is declared until the end of the block, and
   b. By those blocks that are nested within that block if the nested block does not have an identifier with the same name as that of the outside block (the block that encloses the nested block).
3. The scope of a function name is similar to the scope of an identifier declared outside any block. That is, the scope of a function name is the same as the scope of a global variable.

Scope of an Identifier (continued)

- C++ provides a way to access a global variable declared after the definition of a function
  - In this case, the function must not contain any identifier with the same name as the global variable

```
#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
        .
        .
        .
}
```

TABLE 6-2 Scope (Visibility) of the Identifiers

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

```
//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
}
```

Global Variables, Named Constants, and Side Effects

- Using global variables has side effects
- A function that uses global variables is not independent
- If more than one function uses the same global variable and something goes wrong
  - It is difficult to find what went wrong and where
  - Problems caused in one area of the program may appear to be from another area
- Global named constants have no side effects

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
  - Variables declared outside of any block are static variables
  - Declare a static variable within a block by using the reserved word `static`

Static and Automatic Variables (continued)

- The syntax for declaring a static variable is:

```
static dataType identifier;
```

- The statement

```
static int x;
```

  declares x to be a static variable of the type `int`

- Static variables declared within a block are local to the block
  - Their scope is the same as any other local identifier of 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
```

Function Overloading: An Introduction

- In a C++ program, several functions can have the same name
  - This is called function overloading or overloading a function name

Function Overloading (continued)

- Two functions are said to have different formal parameter lists if both functions have:
  - A different number of formal parameters, or
  - If the number of formal parameters is the same, then the data type of the formal parameters, in the order you list them, must differ in at least one position

Function Overloading (continued)

- The following functions all have different formal parameter lists:

```
void functionOne(int x)
void functionTwo(int x, double y)
void functionThree(double y, int x)
int functionFour(char ch, int x, double y)
int functionFive(char ch, int x, string name)
```

- The following functions have the same formal parameter list:

```
void functionSix(int x, double y, char ch)
void functionSeven(int one, double u, char firstCh)
```

Function Overloading (continued)

- Function overloading: creating several functions with the same name
- The signature of a function consists of the function name and its formal parameter list
- Two functions have different signatures if they have either different names or different formal parameter lists
- Note that the signature of a function does not include the return type of the function

Function Overloading (continued)

- Correct function overloading:

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

- Syntax error:

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

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
- You specify the value of a default parameter when the function name appears for the first time (e.g., in the prototype)
- If you do not specify the value of a default parameter, the default value is used

Functions with Default Parameters (continued)

- All default parameters must be the rightmost parameters of the function
- In a function call where the function has more than one default parameter and a value to a default parameter is not specified:
  - You must omit all of the arguments to its right
- Default values can be constants, global variables, or function calls
  - However, you cannot assign a constant value as a default value to a reference parameter

Functions with Default Parameters (continued)

- Consider the following prototype:

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

- Assume:
  - a, b are `int`, ch is `char`, d is `double`
- Examples of legal calls:

```
funcExp(a, b, d);
funcExp(a, 15, 34.6, 'B', 87, ch);
funcExp(b, a, 14.56, 'D');
```

- Examples of illegal calls:

```
funcExp(a, 15, 34.6, 46.7);
funcExp(b, 25, 48.76, 'D', 4567, 78.34);
```

Functions with Default Parameters (continued)

- Examples of illegal function prototypes with default parameters:

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

Programming Example: Classify Numbers

- In this example, we use functions to rewrite the program that determines the number of odds and evens from a given list of integers
- Main algorithm remains the same:
  - Initialize variables, zeros, odds, evens to 0
  - Read a number
  - If number is even, increment the even count
    - If number is also zero, increment the zero count; else increment the odd count
  - Repeat Steps 2-3 for each number in the list

Programming Example: Classify Numbers (continued)

- The program functions include:
  - `initialize`: initialize the variables, such as zeros, odds, and evens
  - `getNumber`: get the number
  - `classifyNumber`: determine if number is odd or even (and whether it is also zero); this function also increments the appropriate count
  - `printResults`: print the results

```
void initialize(int& zeroCount, int& oddCount, int& evenCount)
{
    zeroCount = 0;
    oddCount = 0;
    evenCount = 0;
}
```

```
void getNumber(int& num)
{
    cin >> num;
}
```

```
void classifyNumber(int num, int& zeroCount, int& oddCount,
                    int& evenCount)
{
    switch (num % 2)
    {
    case 0:
        evenCount++;
        if (num == 0)
            zeroCount++;
        break;
    case 1:
    case -1:
        oddCount++;
    } //end switch
} //end classifyNumber
```

```
void printResults(int zeroCount, int oddCount, int evenCount)
{
    cout << "There are " << evenCount << " evens, "
         << "which includes " << zeroCount << " zeros"
         << endl;

    cout << "The number of odd numbers is: " << oddCount
         << endl;
} //end printResults
```

Programming Example: Main Algorithm

- Call `initialize` to initialize variables
- Prompt the user to enter 20 numbers
- For each number in the list
  - Call `getNumber` to read a number
  - Output the number
  - Call `classifyNumber` to classify the number and increment the appropriate count
- Call `printResults` to print the final results

```
int main()
{
        //Variable declaration
    int counter; //loop control variable
    int number;  //variable to store the new number
    int zeros;   //variable to store the number of zeros
    int odds;    //variable to store the number of odd integers
    int evens;   //variable to store the number of even integers

    initialize(zeros, odds, evens);                    //Step 1

    cout << "Please enter " << N << " integers."
         << endl;                                      //Step 2
    cout << "The numbers you entered are: "
         << endl;

    for (counter = 1; counter <= N; counter++)         //Step 3
    {
        getNumber(number);                             //Step 3a
        cout << number << " ";                         //Step 3b
        classifyNumber(number, zeros, odds, evens); //Step 3c
    } // end for loop

    cout << endl;

    printResults(zeros, odds, evens);                  //Step 4

    return 0;
}
```

Summary

- Void function: does not have a data type
  - A `return` statement without any value can be used in a void function to exit it early
  - The heading starts with the word `void`
  - To call the function, you use the function name together with the actual parameters in a stand-alone statement
- Two types of formal parameters:
  - Value parameters
  - Reference parameters

Summary (continued)

- A value parameter receives a copy of its corresponding actual parameter
- A reference parameter receives the memory address of its corresponding actual parameter
  - If a formal parameter needs to change the value of an actual parameter, you must declare this formal parameter as a reference parameter in the function heading

Summary (continued)

- Variables declared within a function (or block) are called local variables
- 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