Programming Fundamentals

Aamina Batool

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

void Functions (Continued)

- The program execution always begins with the first statement in 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
- The `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

The formal parameter list has the following syntax:

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

FORMAL PARAMETER LIST

The formal parameter list has the following syntax:

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

FUNCTION CALL

The function call has the following syntax:

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

ACTUAL PARAMETER LIST

The actual parameter list has the following syntax:

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

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)

- A formal parameter 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 address of the corresponding actual parameter
- A reference parameter stores the address of the corresponding actual parameter

Reference Variables as Parameters (continued)

- During program execution to manipulate the data
  - The address stored in the reference parameter directs it to the memory space of the corresponding actual parameter

Reference Variables as Parameters (continued)

- A reference parameter receives the address of the actual parameter
- Reference parameters can:
  - Pass one or more values from a function
  - Change the value of the actual parameter

Reference Variables as Parameters (continued)

- 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

Parameters & 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

Parameters & Memory Allocation (continued)

- In the case of a reference parameter
  - The address of the actual parameter passes to the formal parameter
- Content of the formal parameter is an address

Parameters & Memory Allocation (continued)

- During execution, changes made by the formal parameter permanently change the value of the actual parameter

EXAMPLE 7-7

The following program shows how reference and value parameters work.

```
//Example 7-7: Reference and value parameters

#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
```

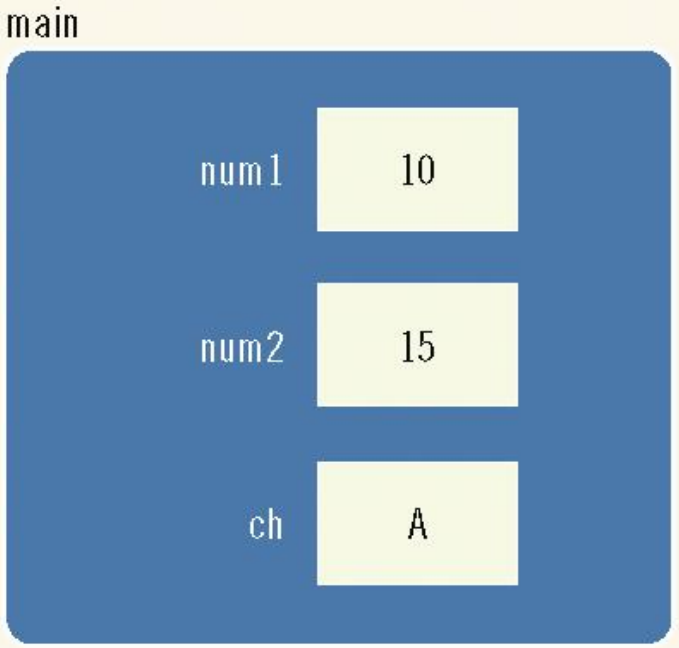

FIGURE 7-5 Values of the variables after the statement in Line 3 executes

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

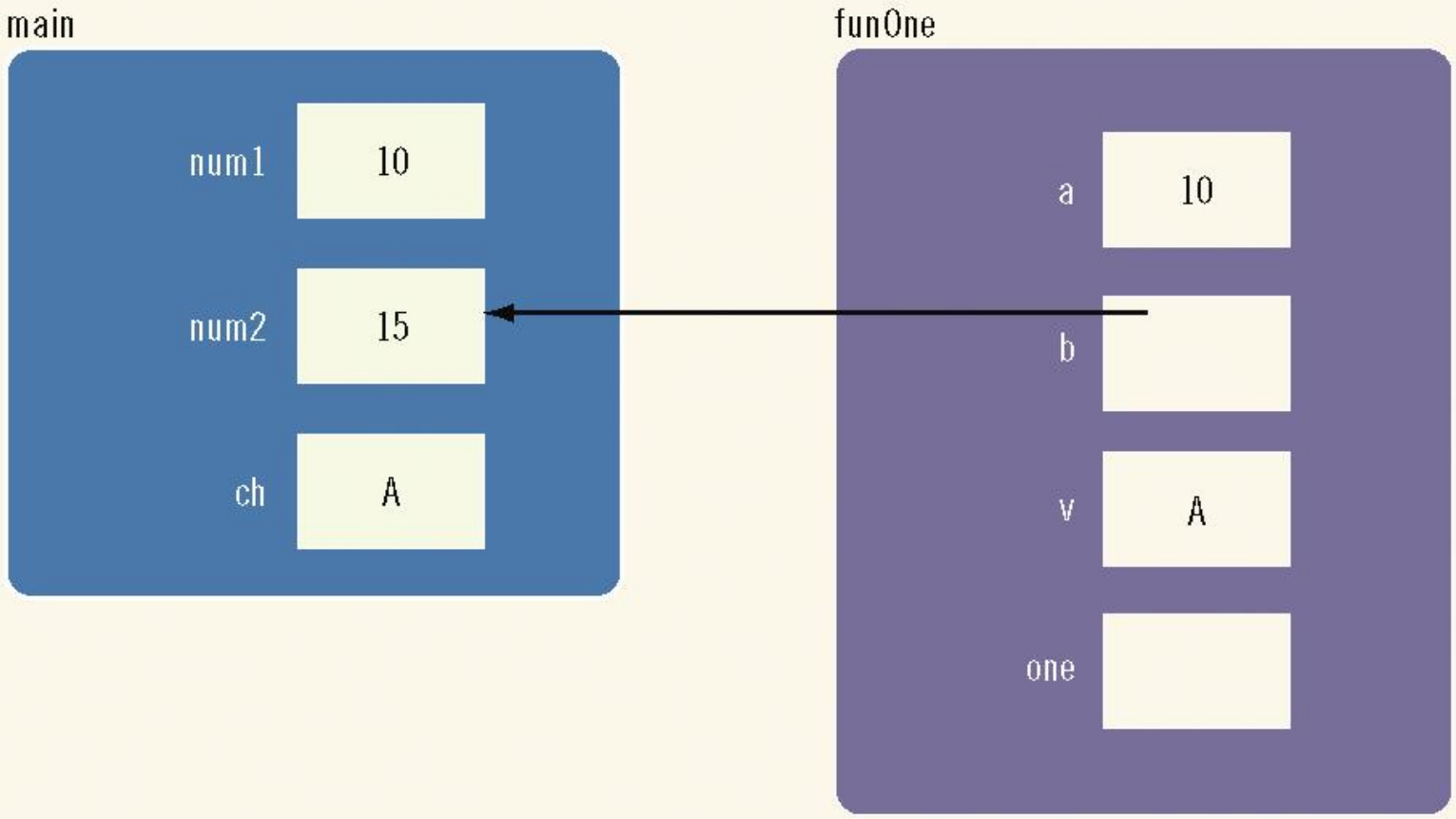

FIGURE 7-6 Values of the variables just before the statement in Line 9 executes

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

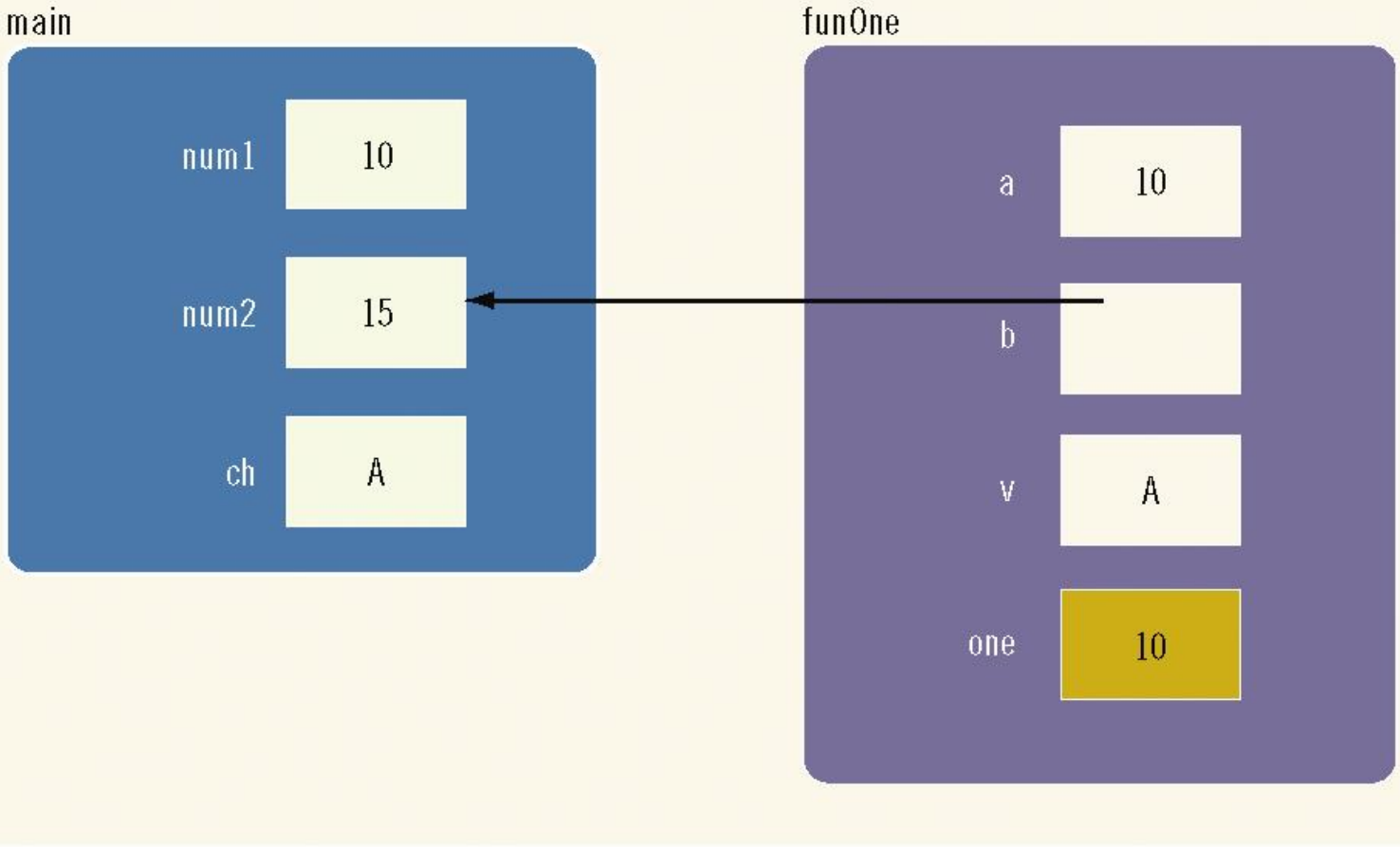

FIGURE 7-7 Values of the variables after the statement in Line 9 executes

```
a++;                                                    //Line 10
```

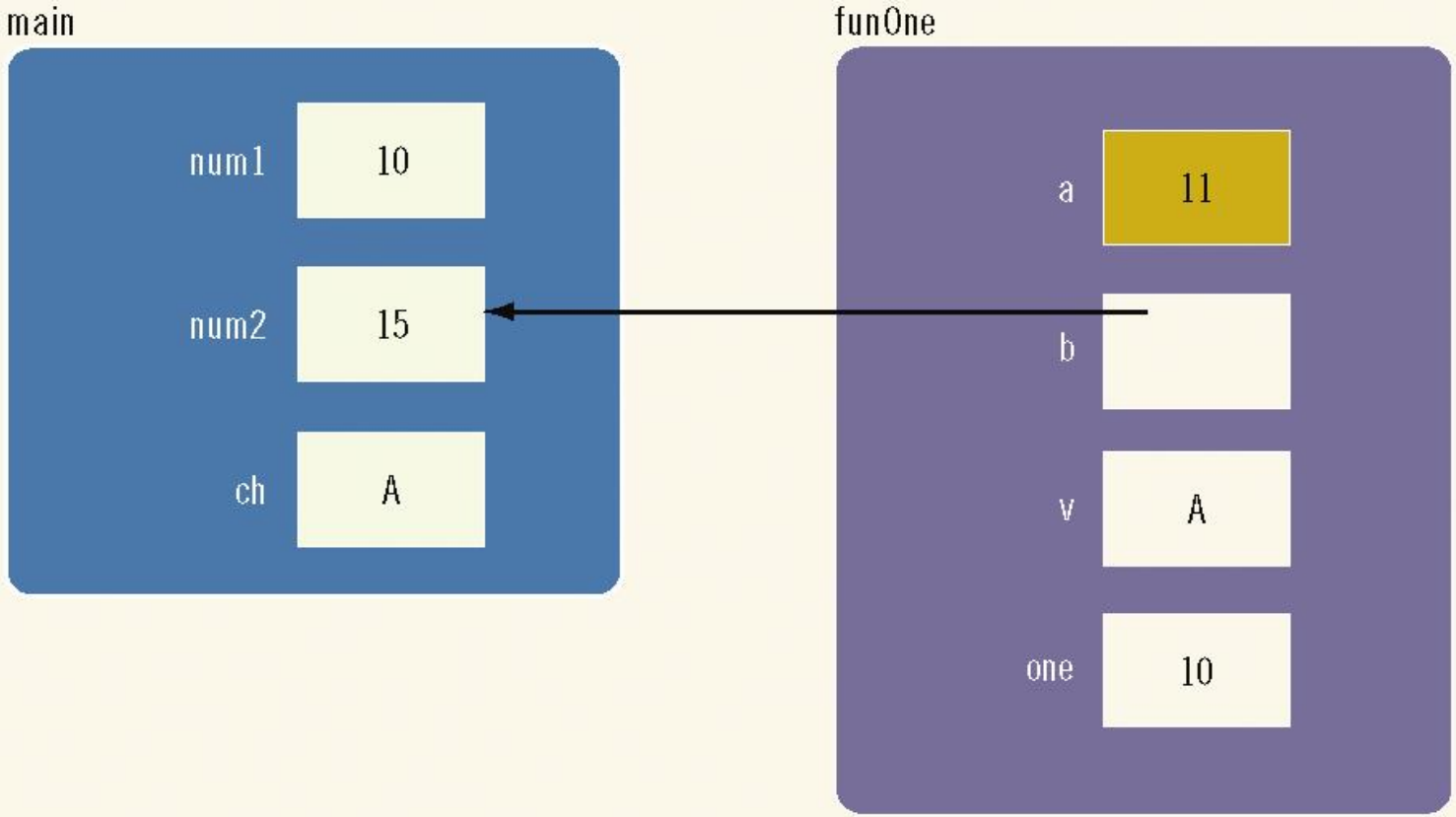

FIGURE 7-8 Values of the variables after the statement in Line 10 executes

```
b = b * 2;                                    //Line 11
```

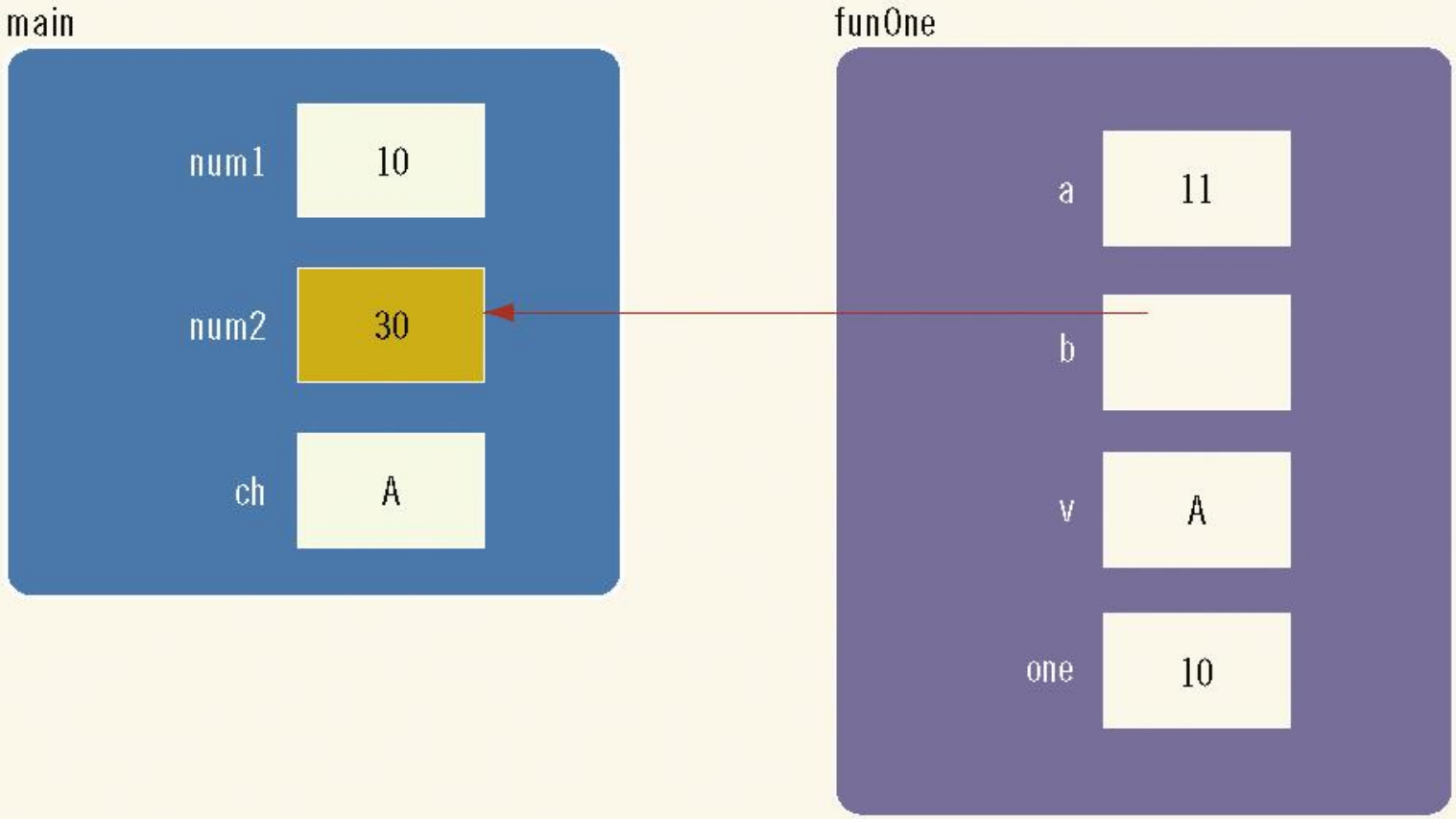

FIGURE 7-9 Values of the variables after the statement in Line 11 executes

```
v = 'B';                                                    //Line 12
```

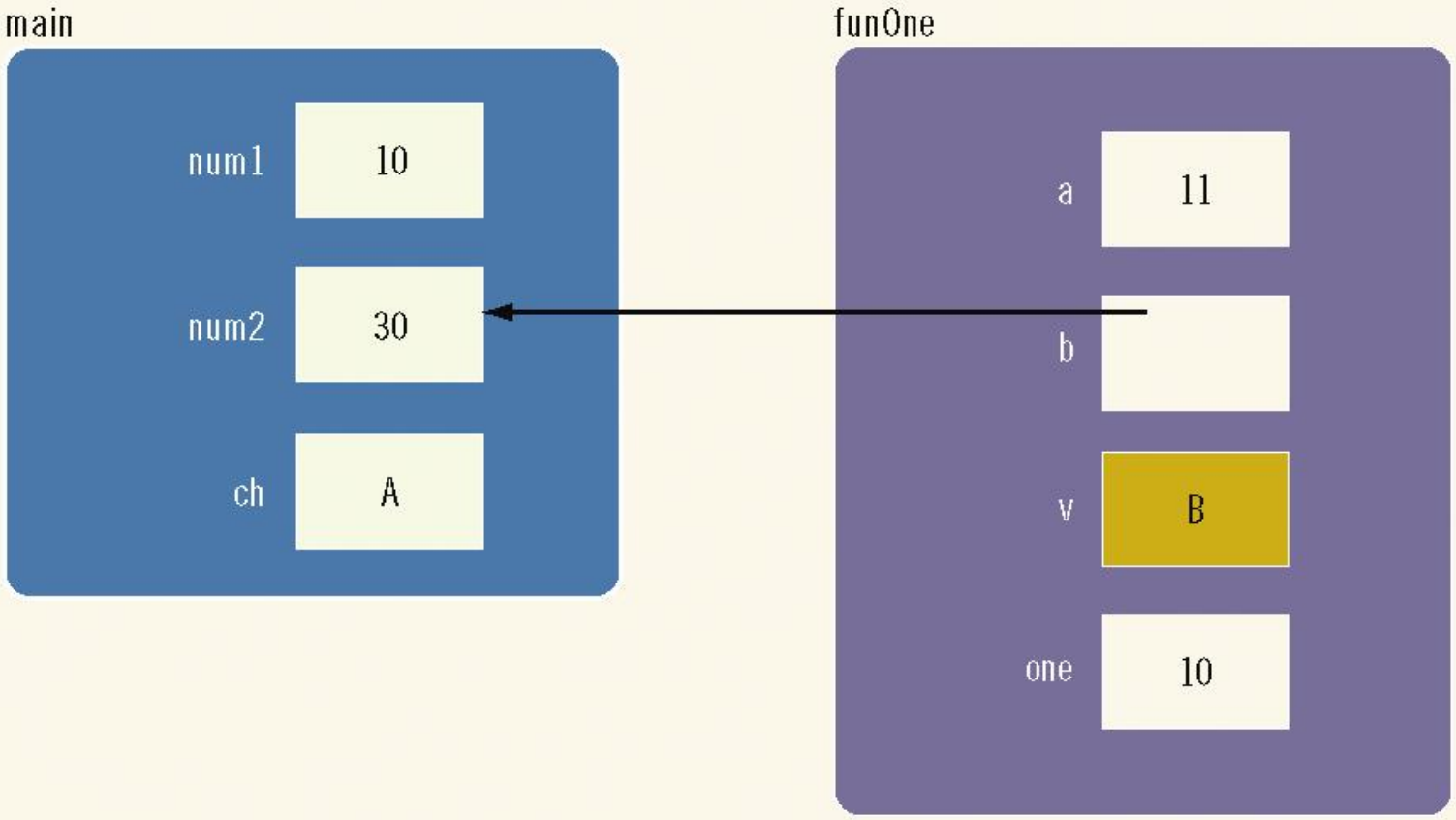

FIGURE 7-10 Values of the variables after the statement in Line 12 executes

```
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
```

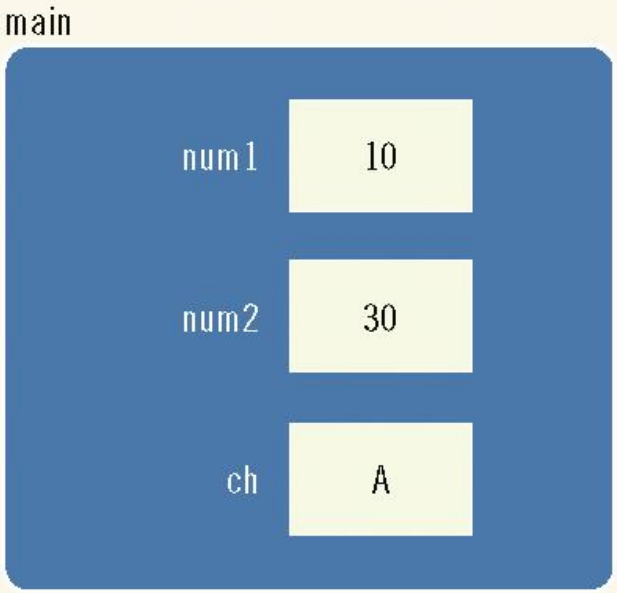

FIGURE 7-11 Values of the variables when control goes back to Line 6

Line 6 produces the following output:

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

```
x++;                                                    //Line 14
```

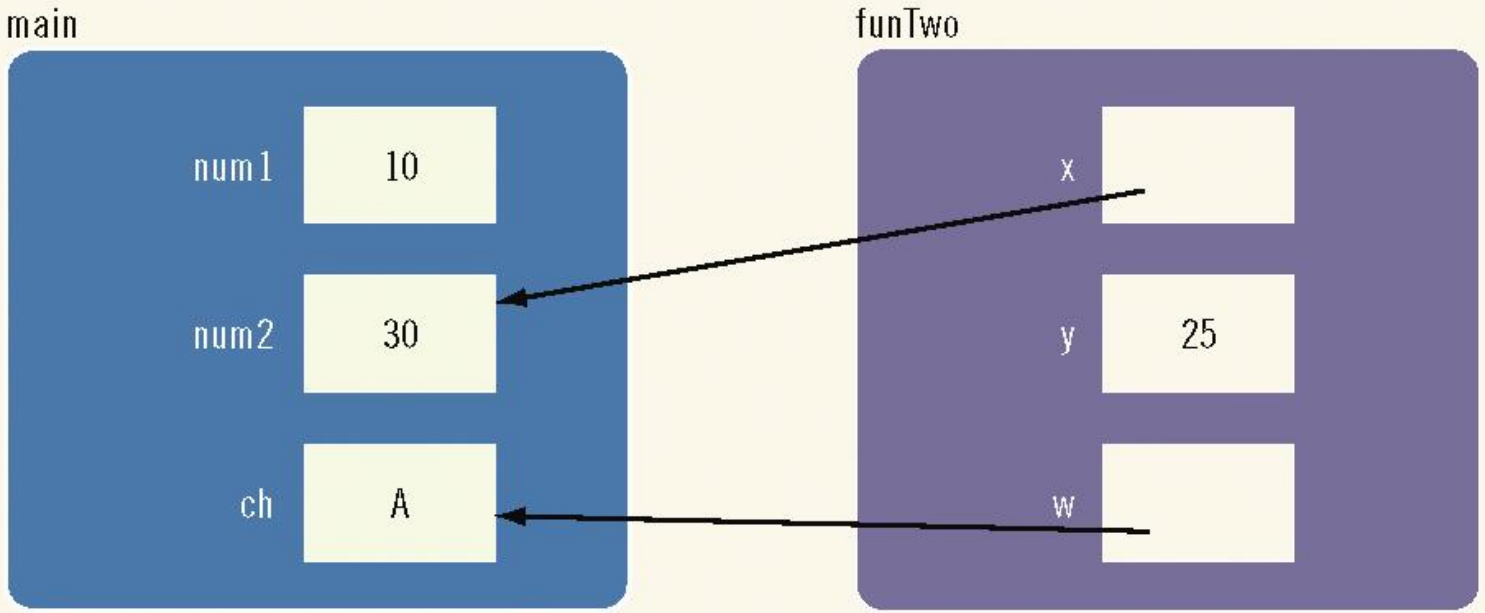

FIGURE 7-12 Values of the variables before the statement in Line 14 executes

```
x++;                                                    //Line 14
```

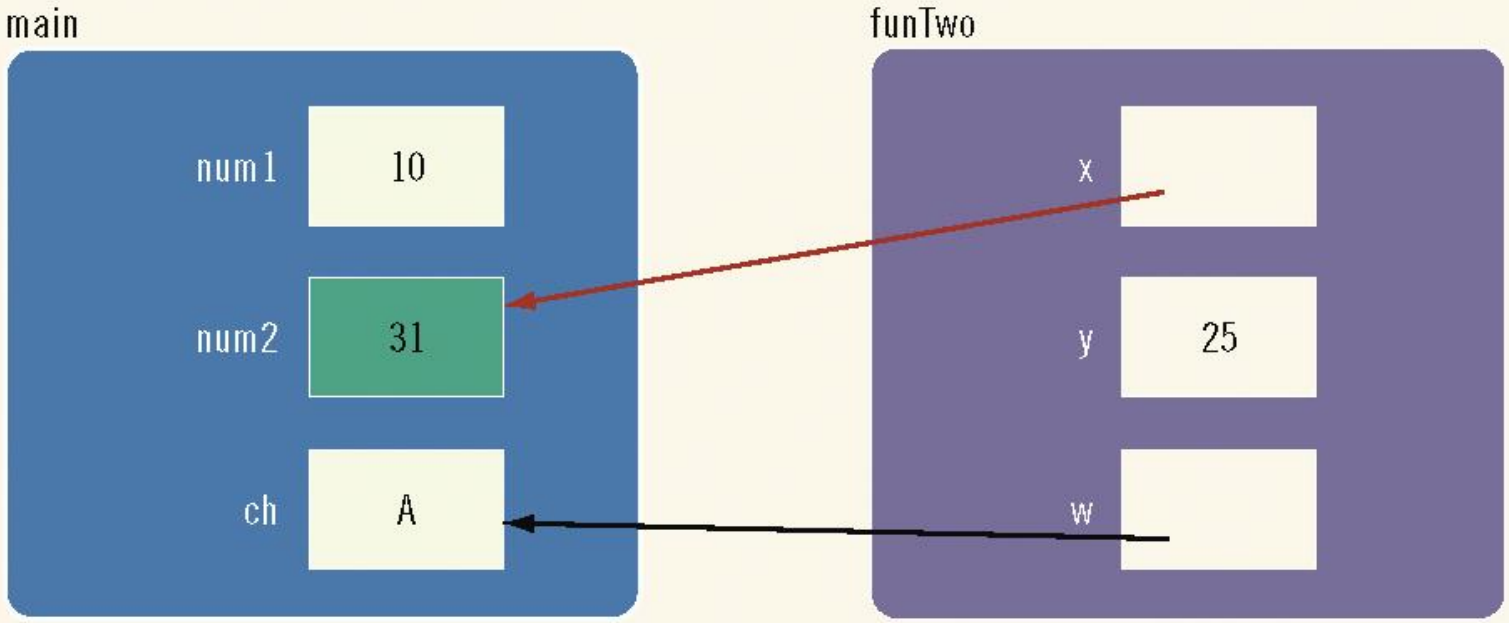

FIGURE 7-13 Values of the variables after the statement in Line 14 executes

```
y = y * 2;                                                    //Line 15
```

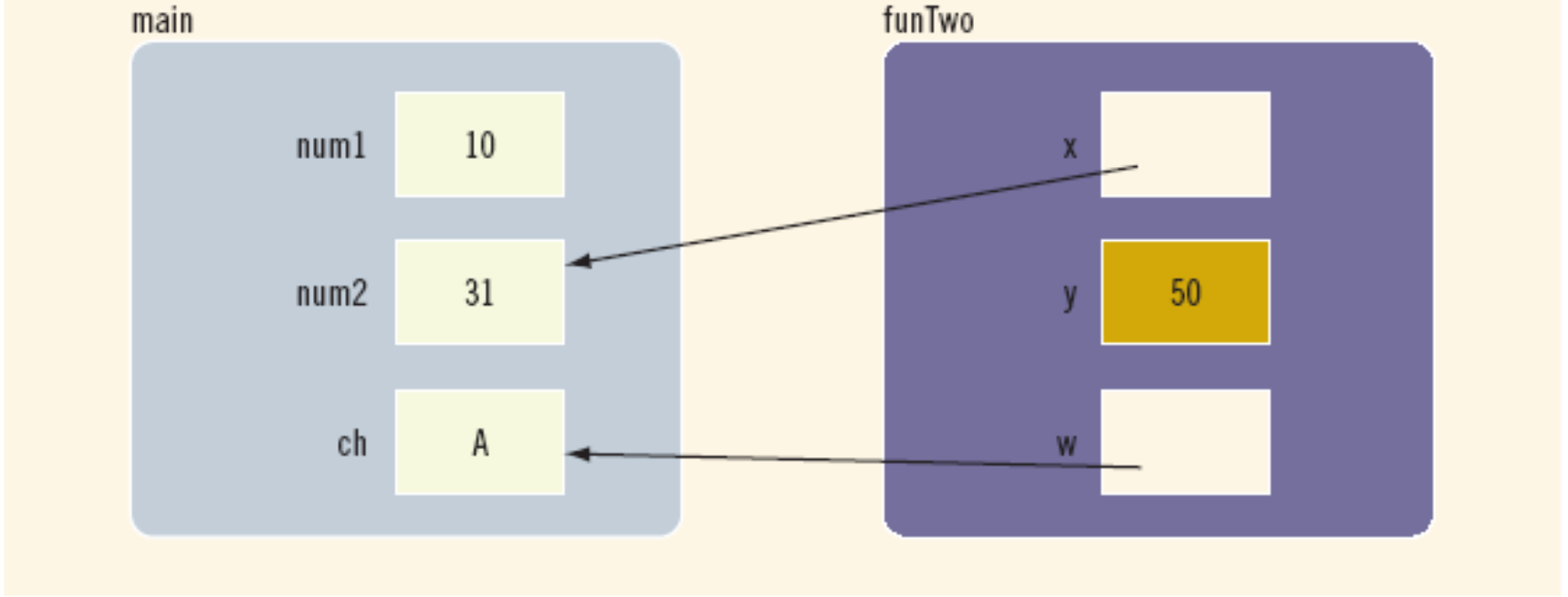

FIGURE 7-14 Values of the variables after the statement in Line 15 executes

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

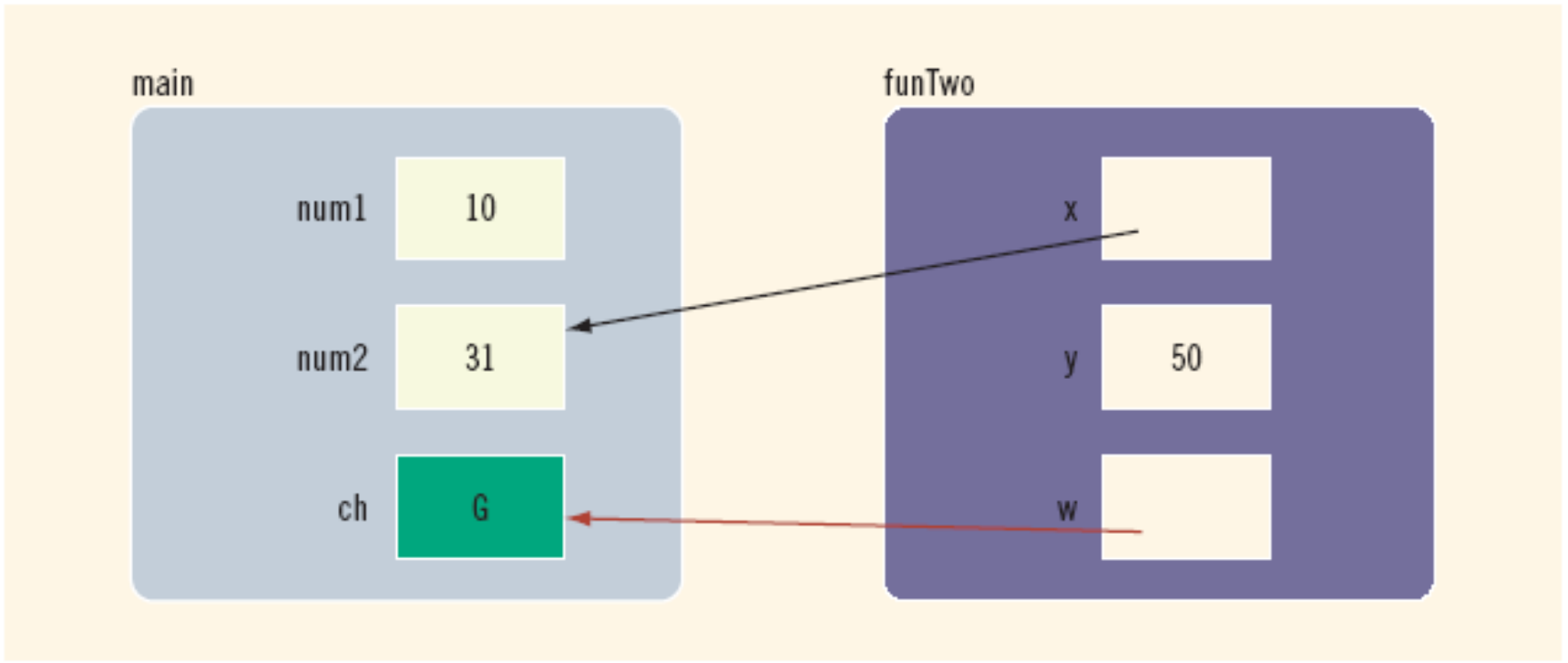

FIGURE 7-15 Values of the variables after the statement in Line 16 executes

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

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
```

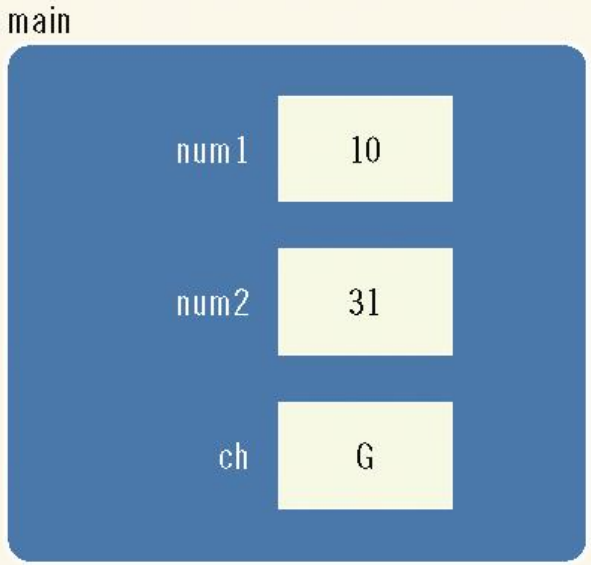

FIGURE 7-16 Values of the variables when control goes to Line 8

The statement in Line 8 produces the following output:

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

Static and Automatic Variables

- Automatic variable - memory is allocated at function entry and deallocated at function exit
- Static variable - memory remains allocated as long as the program executes
- Variables declared outside of any function are static variables
- By default, variables declared within a function are automatic variables
- You can declare a static variable within a function 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

Global Variables

- The operator :: is called the scope resolution operator
- By using the scope resolution operator
  - A global variable declared before the definition of a function (block) can be accessed by the function (or block) even if the function (or block) has an identifier with the same name as the variable

```cpp
#include <iostream>
using namespace std;

// defining the global variable
int a=10;

int main()
{
    //local variable
    int a=15;
    cout<<"local a: "<<a<<" Global a: "<<::a;

    // Re-defining global variable by using ::
    ::a=20;
    cout<<"\nlocal a: "<<a<<" Global a: "<<::a;
    return 0;
}
```

Function Prototype

- Function prototype: function heading without the body of the function
- Syntax:

```
functionType functionName(parameter list);
```

- It is not necessary to specify the variable name in the parameter list
- The data type of each parameter must be specified

Function Overloading

- 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
- void fun(int, int);
- void fun(int, double);

```
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)
```

These functions all have different formal parameter lists.

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

The functions `functionSix` and `functionSeven` both have three formal parameters and the data type of the corresponding parameters is the same; therefore, these functions have the same formal parameter list

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

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

These function headings correctly overload the function `functionXYZ`:

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

- Both of these function headings have the same name and same formal parameter list
- Therefore, these function headings to overload the function `functionABC` are incorrect
- In this case, the compiler will generate a syntax error
- Note that the return types of these function headings are different

Functions with Default Parameters

- When a function is called
  - 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, such as in the prototype

Functions with Default Parameters (continued)

- If you do not specify the value of a default parameter
  - The default value is used
- All of the 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

Functions with Default Parameters (continued)

- Default values can be constants, global variables, or function calls
- The caller has the option of specifying a value other than the default for any default parameter
- You cannot assign a constant value as a default value to a reference parameter

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

```
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');`

The following function calls are illegal:

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

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);`

References

1. C++ Programming: From Problem Analysis to Program Design, Third Edition
2. https://www.just.edu.jo/~yahya-t/cs115/