CS1002 – Programming Fundamentals

Lecture # 23
Monday, November 21, 2022
FALL 2022
FAST – NUCES, Faisalabad Campus

Rizwan Ul Haq

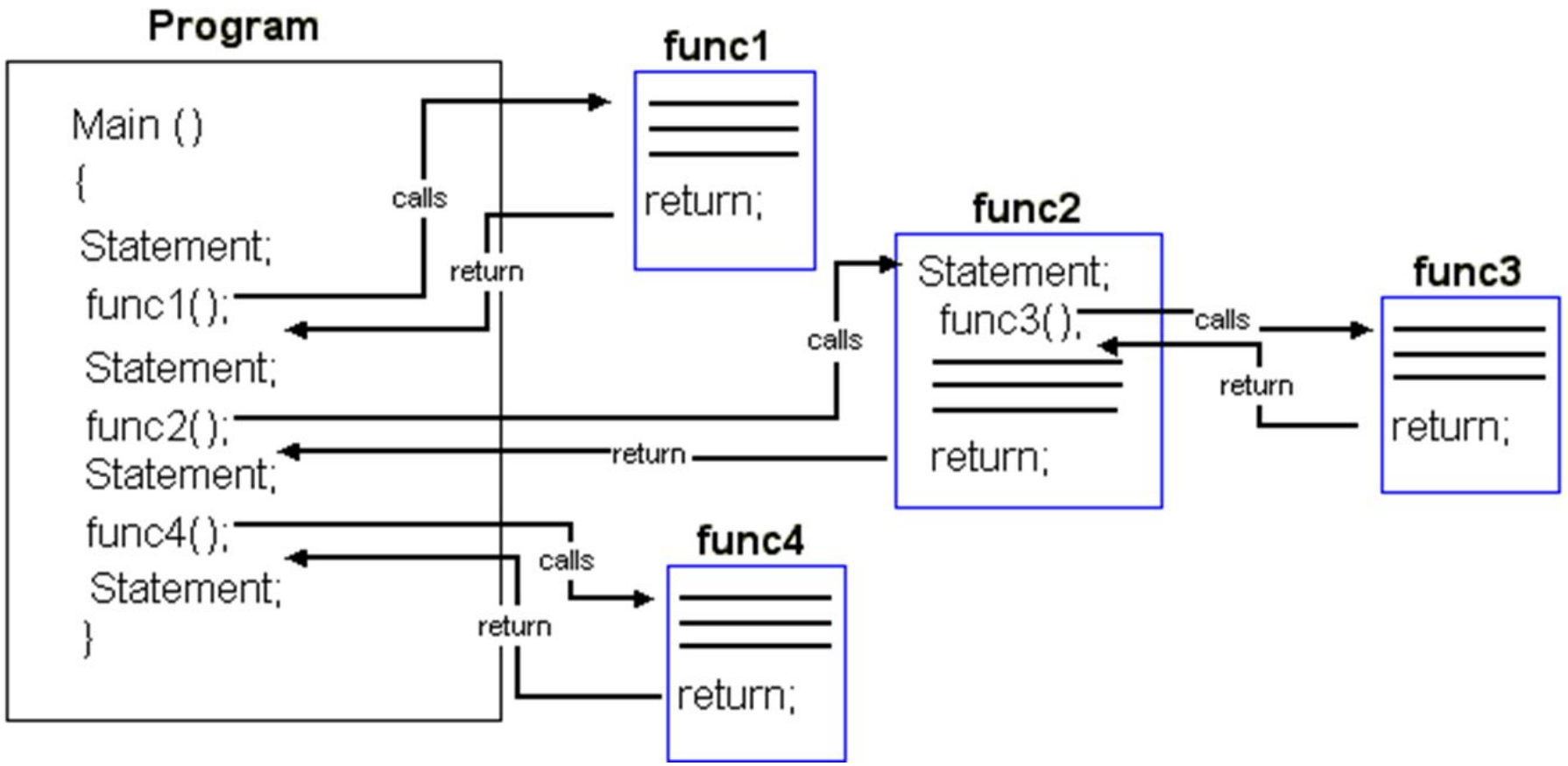
Flow of Execution

- Execution always begins at the first statement in the function **main**
- Other functions are executed **only when they are called**
- Function prototypes appear **before any function definition**
  - The compiler translates these **first**
- The compiler can then correctly translate a function call

Flow of Execution (cont'd.)

- A function call results in **transfer of control** to the first statement in the body of the called function
- After the last statement of a function is executed, **control is passed** back to the point immediately following the function call
- A value-returning function returns a value
  - After executing the function the returned value replaces the function call statement

Conti…


Program
Main ()
{
Statement;
func1();
Statement;
func2();
Statement;
func4();
Statement;
}
calls
return
calls
return
func1
return;
func2
Statement;
func3();
return;
calls
return
func3
return;
calls
return
func4
return;

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 (cont'd.)**

- A void function does not have a return type
  - **return** statement without any value is typically used to exit the function **early**.
  - **It can be used** in void **for early termination**
- Formal parameters are **optional**
- A call to a void function is a stand-alone statement

void **Functions (cont'd.)**

- Function definition syntax:

```
void functionName(formal parameter list)
{
    statement(s)
}
```

- Formal parameter list syntax:

```
dataType variable, dateType variable, . . .
```

void **Functions (cont'd.)**

- Function call syntax:

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

- Actual parameter list syntax:

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

Reference variable

- A reference variable is an alias to an already declared variable
- It does not occupy any memory
- It needs to be initialized by the address of any variable at the time of declaration

```
int x = 25;
int &y = x;
```

- The above code declares a variable x and a reference variable y that refers to x
- If we access y in-fact we'll be accessing x

```
cin >> y;  //This will read the value from user into y assume 55
cout << x; //This will print 55 as the value is stores in x
           //through its reference y
```

void **Functions (cont'd.)**

- **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

void Functions (cont'd.)

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

This function funExp has four parameters.

void Functions (cont'd.)

```
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`
4. **four**, a reference parameter of type `double`

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**

Example

```
void funcValueParam(int num);

int main()
{
    int number = 6;                                          //Line 1
    cout << "Line 2: Before calling the function"
         << "funcValueParam, number = " << number << endl;   //Line 2

    funcValueParam(number);                                  //Line 3

    cout << "Line 4: After calling the function"
         << "funcValueParam, number = " << number << endl;   //Line 4
    return 0;
}
void funcValueParam(int num)
{
    cout << "Line 5: In the function funcValueParam, "
         << "before changing, num = " << num     << endl;    //Line 5
    num = 15;                                                //Line 6
    cout << "Line 7: In the function funcValueParam, "
         << "after changing, num = " << num << endl;         //Line 7
}
```

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 Benefits

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

Example 7-5: 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 cScore);

int main()
{
    int courseScore;
    cout << "Line 1: Based on the course score, \n"
        << "\tthis program computes the "
        << "course grade." << endl;          //Line 1

    getScore(courseScore);                   //Line 2
    printGrade(courseScore);                 //Line 3

    return 0;
}
```

Example 7-5: Calculate Grade (cont'd.)

```
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;
}
```

Example 7-5: Calculate Grade (cont'd.)

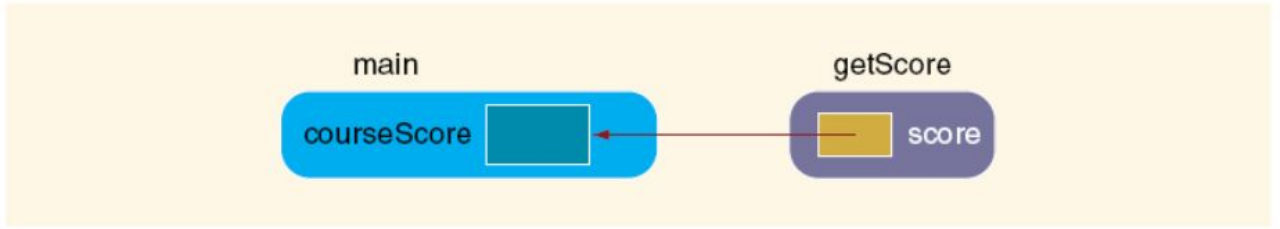


FIGURE 7-1 Variable `courseScore` and the parameter `score`

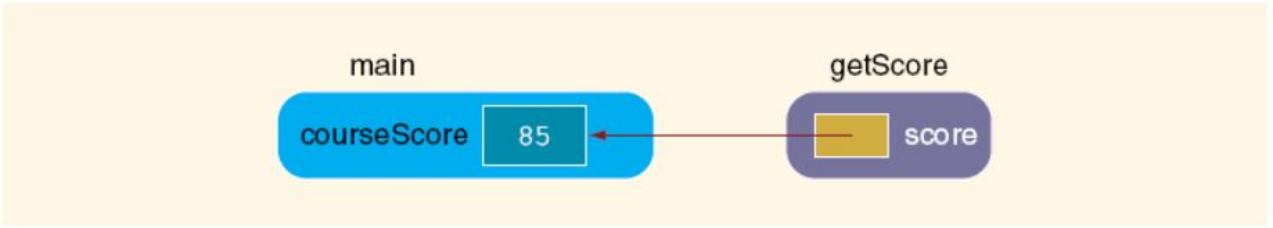


FIGURE 7-2 Variable `courseScore` and the parameter `score` after the statement in Line 5 executes

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

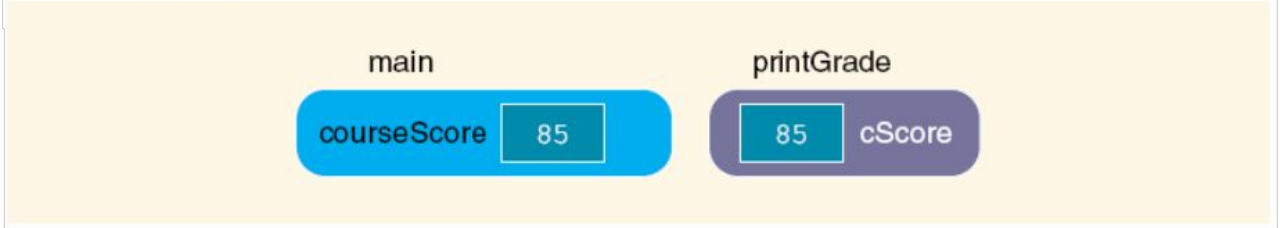


FIGURE 7-4 Variable `courseScore` and the parameter `cScore`

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 (cont'd.)

- 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

Value and Reference Parameters and Memory Allocation (cont'd.)

```
// This following program shows how reference and value parameter work
//Example 7-6: Reference and Value parameter
#include <iostream>
using namespace std;
void funOne(int a, int& b, char c);
void funTwo(int& x, int y, char& w);
int main(){
    int num1 = 10, num2 = 15;
    char ch = 'A';
    cout << "Line 4: Inside main: num1 = " << num1
        << ", num2 = " << num2 << ", ch = "
        << ch << endl;              //Line 4
    funOne(num1, num2, ch);         //Line 5
    cout << "Line 6: Inside main After funOne: num1 = " << num1
        << ", num2 = " << num2 << ", ch = "
        << ch << endl;              //Line 6
    funTwo(num2, 25, ch);           //Line 7
    cout << "Line 8: Inside main After funOne: num1 = " << num1
        << ", num2 = " << num2 << ", ch = "
        << ch << endl;              //Line 8
    return 0;
}
```

Value and Reference Parameters and Memory Allocation (cont'd.)

```
void funOne(int a, int& b, char c)
{
    int one;
    one = a;              //Line 9
    a++;                  //Line 10
    b = b * 2;            //Line 11
    c = 'B';              //Line 12

    cout << "Line 13: Inside funOne: a = " << a
         << ", b = " << b << ", c = " << c
         << ", 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
}
```

Value and Reference Parameters and Memory Allocation (cont'd.)

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
```

Value and Reference Parameters and Memory Allocation (cont'd.)

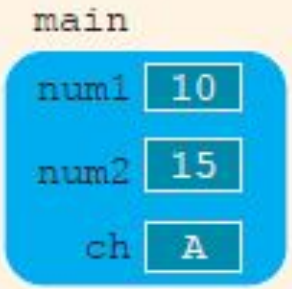


FIGURE 7-5 Values of the variables after the statement in Line 3 executes

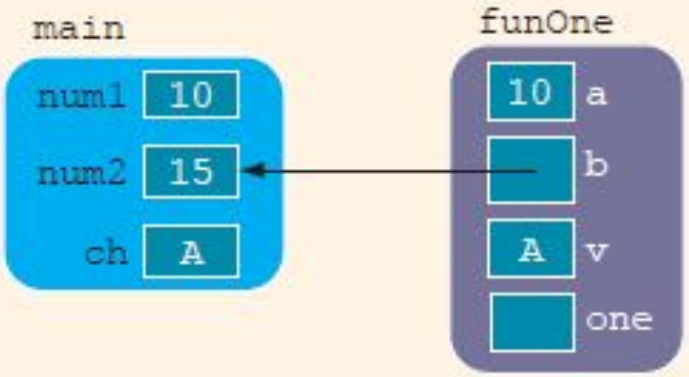


FIGURE 7-6 Values of the variables just before the statement in Line 9 executes

Value and Reference Parameters and Memory Allocation (cont'd.)

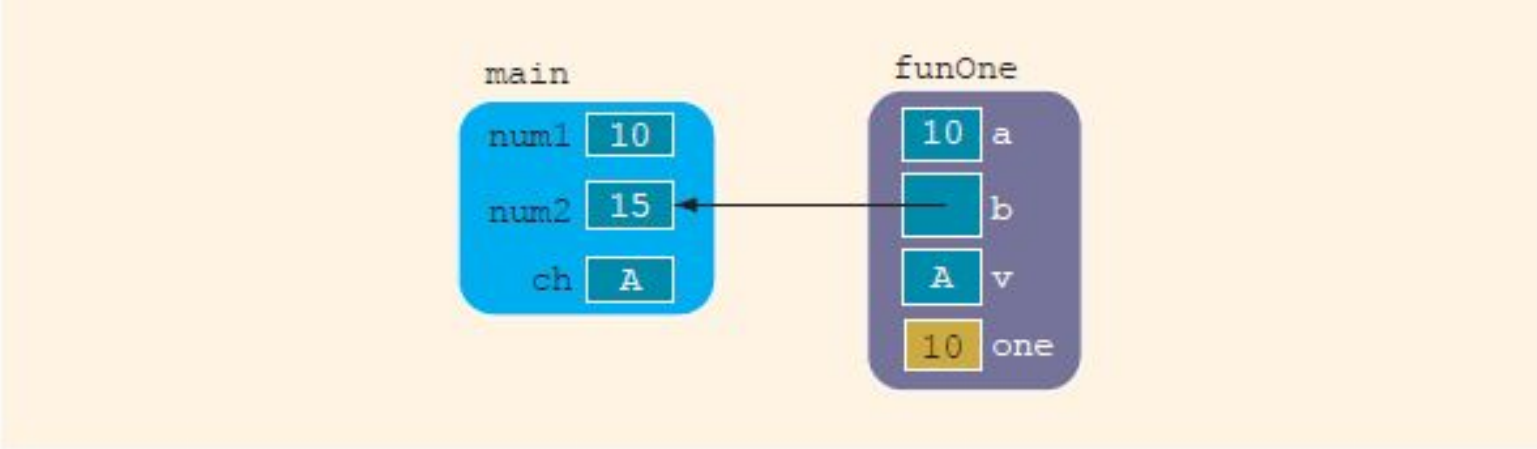


FIGURE 7-7 Values of the variables after the statement in Line 9 executes

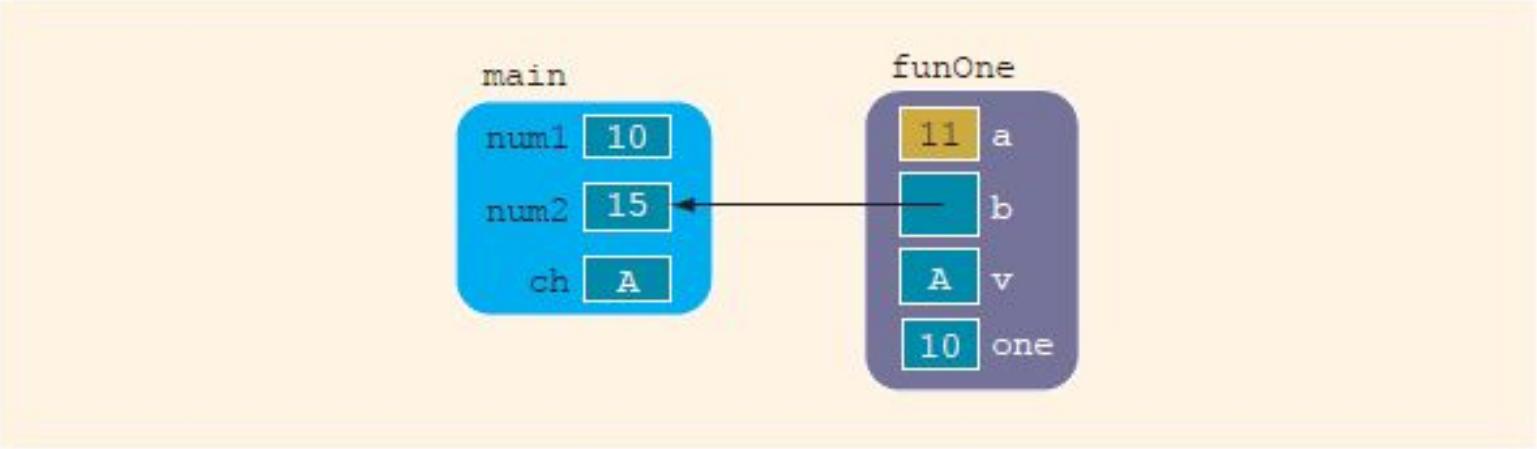


FIGURE 7-8 Values of the variables after the statement in Line 10 executes

Value and Reference Parameters and Memory Allocation (cont'd.)

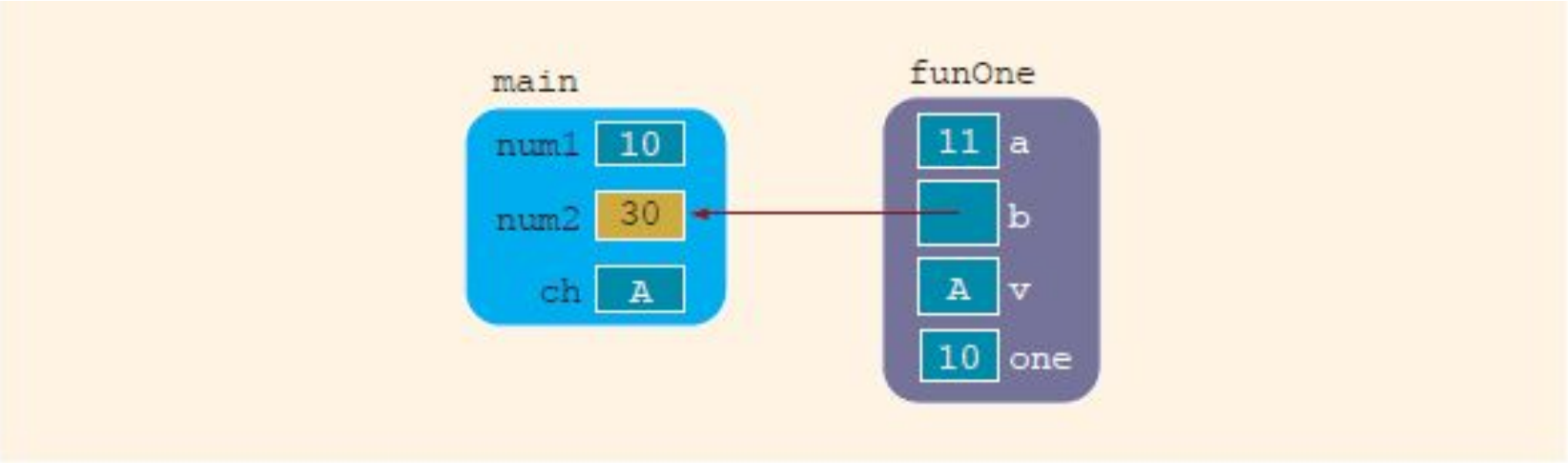


FIGURE 7-9 Values of the variables after the statement in Line 11 executes

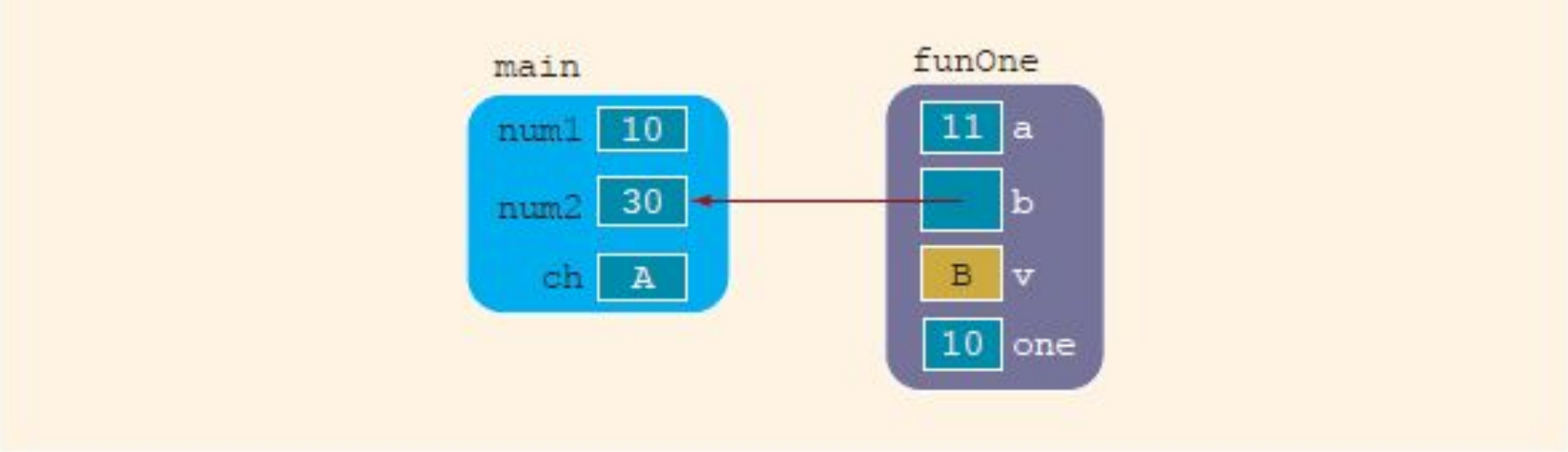


FIGURE 7-10 Values of the variables after the statement in Line 12 executes

Value and Reference Parameters and Memory Allocation (cont'd.)

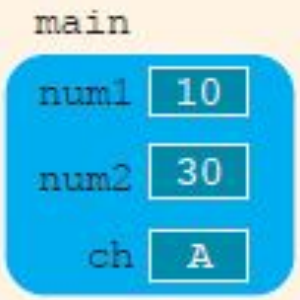


FIGURE 7-11 Values of the variables when control goes back to Line 6

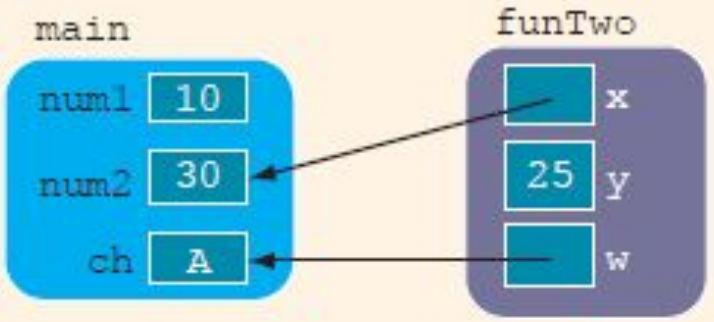


FIGURE 7-12 Values of the variables before the statement in Line 14 executes

Value and Reference Parameters and Memory Allocation (cont'd.)

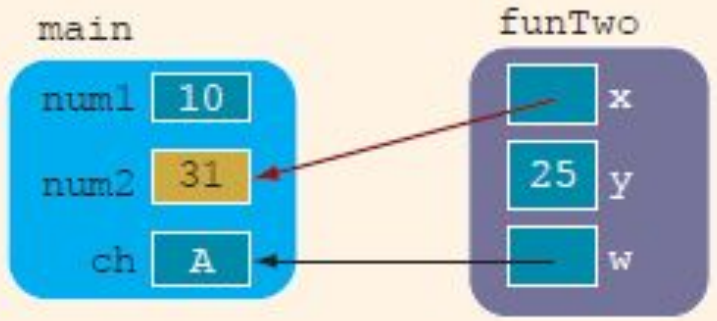


FIGURE 7-13 Values of the variables after the statement in Line 14 executes

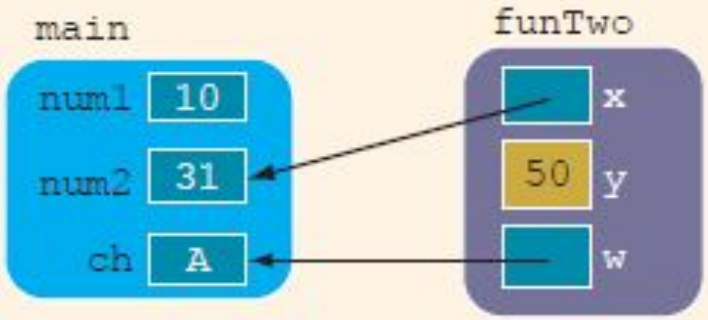


FIGURE 7-14 Values of the variables after the statement in Line 15 executes

Value and Reference Parameters and Memory Allocation (cont'd.)

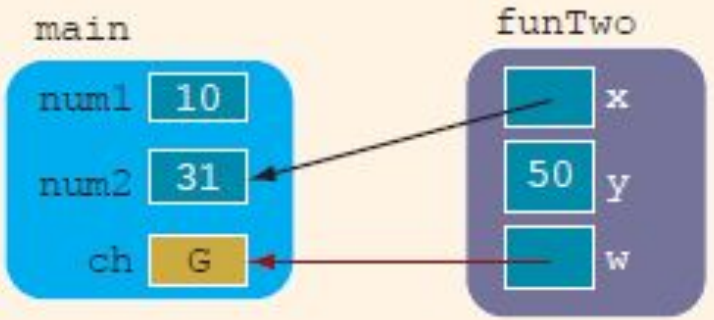


FIGURE 7-15 Values of the variables after the statement in Line 16 executes

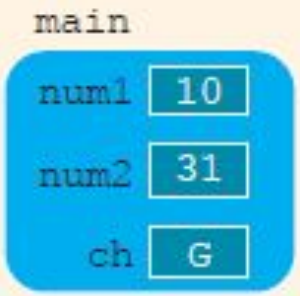


FIGURE 7-16 Values of the variables when control goes to Line 8

```
//Example 7-7: Reference and value parameters.
//Program: Makes you think.

#include <iostream>

using namespace std;

void addFirst(int& first, int& second);
void doubleFirst(int one, int two);
void squareFirst(int& ref, int val);

int main()
{
    int num = 5;

    cout << "Line 1: Inside main: num = " << num
         << endl;                                          //Line 1

    addFirst(num, num);                                    //Line 2
    cout << "Line 3: Inside main after addFirst:"
         << " num = " << num << endl;                      //Line 3

    doubleFirst(num, num);                                 //Line 4
    cout << "Line 5: Inside main after "
         << "doubleFirst: num = " << num << endl;          //Line 5

    squareFirst(num, num);                                 //Line 6
    cout << "Line 7: Inside main after "
         << "squareFirst: num = " << num << endl;          //Line 7

    return 0;
}
```

```
void addFirst(int& first, int& second)
{
    cout << "Line 8: Inside addFirst:  first = "
         << first << ", second = " << second << endl; //Line 8

    first = first + 2;                                 //Line 9

    cout << "Line 10: Inside addFirst:  first = "
         << first << ", second = " << second << endl; //Line 10


    second = second * 2;                               //Line 11

    cout << "Line 12: Inside addFirst:  first = "
         << first << ", second = " << second << endl; //Line 12
}

void doubleFirst(int one, int two)
{
    cout << "Line 13: Inside doubleFirst:  one = "
         << one << ", two = " << two << endl;          //Line 13

    one = one * 2;                                     //Line 14

    cout << "Line 15: Inside doubleFirst:  one = "
         << one << ", two = " << two << endl;          //Line 15

    two = two + 2;                                     //Line 16

    cout << "Line 17: Inside doubleFirst:  one = "
         << one << ", two = " << two << endl;          //Line 17
}
```

```
void squareFirst(int& ref, int val)
{
    cout << "Line 18: Inside squareFirst: ref = "
         << ref << ", val = " << val << endl;          //Line 18

    ref = ref * ref;                                   //Line 19

    cout << "Line 20: Inside squareFirst: ref = "
         << ref << ", val = " << val << endl;          //Line 20

    val = val + 2;                                     //Line 21

    cout << "Line 22: Inside squareFirst: ref = "
         << ref << ", val = " << val << endl;          //Line 22
}
```

Sample Run:

```
Line 1: Inside main:  num = 5
Line 8: Inside addFirst:  first = 5, second = 5
Line 10: Inside addFirst:  first = 7, second = 7
Line 12: Inside addFirst:  first = 14, second = 14
Line 3: Inside main after addFirst:  num = 14
Line 13: Inside doubleFirst:  one = 14, two = 14
Line 15: Inside doubleFirst:  one = 28, two = 14
Line 17: Inside doubleFirst:  one = 28, two = 16
Line 5: Inside main after doubleFirst:  num = 14
Line 18: Inside squareFirst: ref = 14, val = 14
Line 20: Inside squareFirst: ref = 196, val = 14
Line 22: Inside squareFirst: ref = 196, val = 16
Line 7: Inside main after squareFirst:  num = 196
```

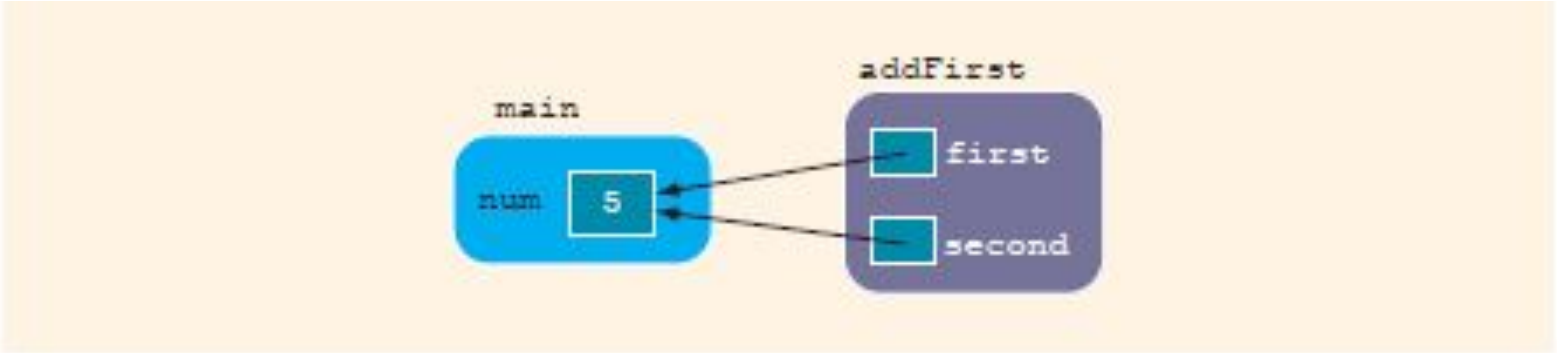


FIGURE 7-17 Parameters of the function addFirst

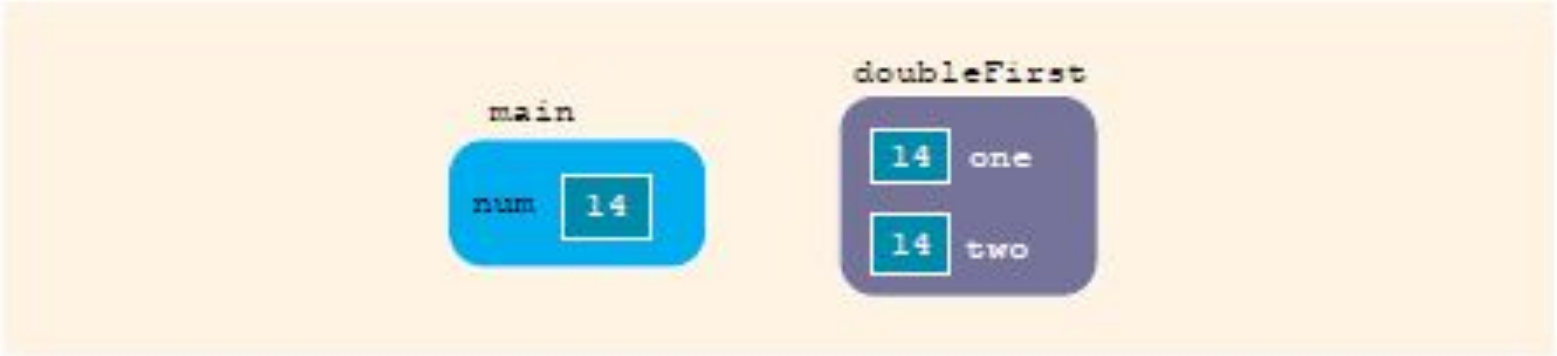


FIGURE 7-18 Parameters of the function doubleFirst

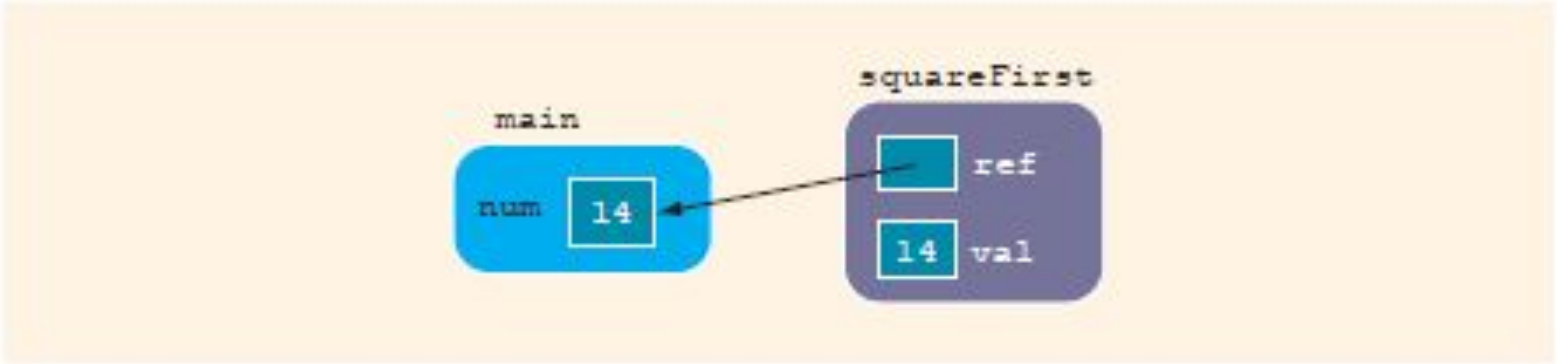


FIGURE 7-19 Parameters of the function squareFirst

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

Using Functions in a Menu-Driven Program

Functions can be used:

- To implement user choices from menu
- To implement general-purpose tasks
  - Higher-level functions can call general-purpose functions
  - This minimizes the total number of functions and speeds program development time

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

```
#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
```

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