IMAM ABDULRAHMAN BIN FAISAL UNIVERSITY
College of Computer Science & IT
Department of CS

Welcome to CS 221:Fundamentals of Programming

Weeks (6): Functions – Part 2

Chapters 9 and 10 (Zak Textbook)

Chapter 4 (Savitch Textbook)

Objectives

- Introduce user\programmer-defined functions and discover how to create them in a program.
- Passing by reference and passing by value
- Variables scope: Local and global

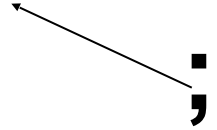
USER\PROGRAMMER-DEFINED FUNCTIONS

What is user-defined functions

- A user-defined function (UDF) is a function provided by the user of a program or environment, in a context where the usual assumption is that functions are built into the program or environment.

Programmer-Defined Functions

- Two components of a function definition
  - Function declaration (or function prototype)
    - Shows how the function is called
    - Must appear in the code before the function can be called
    - **Syntax**:
      Type_returned Function_Name(Parameter_List);



  - Function definition
    - Describes how the function does its task
    - Can appear before or after the function is called
    - Syntax:

```
Type_returned  Function_Name(Parameter_List)
  {
      //code to make the function work
  }
```

Function Declaration

- Tells the return type
- Tells the name of the function
- Tells how many arguments are needed
- Tells the types of the arguments
- Tells the formal parameter names
  - Formal parameters are like placeholders for the actual arguments used when the function is called
  - Formal parameter names can be any valid identifier
- Example:

```
double totalCost(int numberPar, double pricePar);
// Compute total cost including 5% sales tax on
// numberPar items at cost of pricePar each
```

Function Definition Syntax

Syntax for a Function That Returns a Value

Function Declaration

```
Type_Returned   Function_Name(Parameter_List);
Function_Declaration_Comment
```

Function Definition

```
Type_Returned   Function_Name(Parameter_List)
{
    Declaration_1
    Declaration_2
        . . .
    Declaration_Last
    Executable_Statement_1
    Executable_Statement_2
        . . .
    Executable_Statement_Last
}
```

function header

body

Must include one or more `return` *statements.*

Function Definition

- **Function body** contains instructions for performing the function's assigned task
- Surrounded by braces ({ })
- Last statement is usually the **return statement**
  - Returns one value (must match return data type in function header)
- After return statement is processed, program execution continues in calling function

Function Prototypes

- When a function definition appears below the `main` function, you must enter a function prototype above the `main` function
- A **function prototype** is a statement that specifies the function's name, data type of its return value, and data type of each of its formal parameters (if any)
  - ***Names for the formal parameters are not required***
- Programmers usually place function prototypes at beginning of program, after the `#include directives`

Calling a Function

- Like Pre-defined functions, A user-defined function must be called (invoked) to perform its task
- `main` is automatically called when program is run
- Other functions must be called by a statement
- Syntax for calling a function:

  functionName ([*argumentList*]) ;

  - *argumentList* is list of actual arguments (if any)
  - An actual argument can be a variable, named constant, literal constant, or keyword.
- Number, data type, and ordering of actual arguments must match the formal parameters in function header

Calling a Function (cont'd.)

- Value-returning functions are typically called from statements that:
  - Assign the return value to a variable
  - Use the return value in a calculation or comparison
  - Display the return value

Calling a Function (cont'd.)

- C++ allows you to pass either a variable's value or its address to a function
  - Passing a variable's value is referred to as **passing *by value***
  - Passing a variable's address is referred to as **passing *by reference***

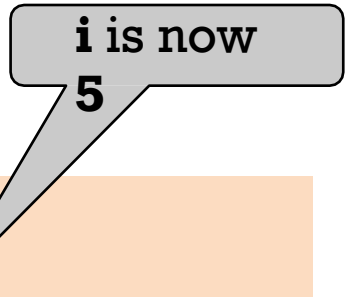
Flow of Execution

- 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 (continued)

- 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

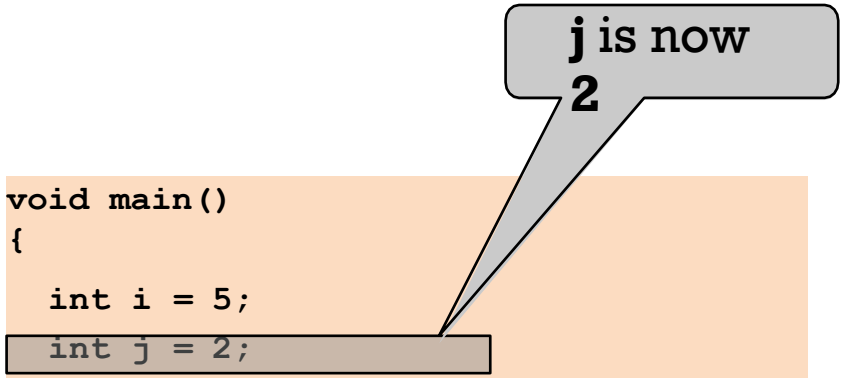
Flow of Execution (continued)




```
void main()
{
  int i = 5;
  int j = 2;
  int k = max( i, j );

cout << "The maximum between " << i <<
   "and " << j << " is " << k;
}
```

```
int max(int num1, int num2}
{
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

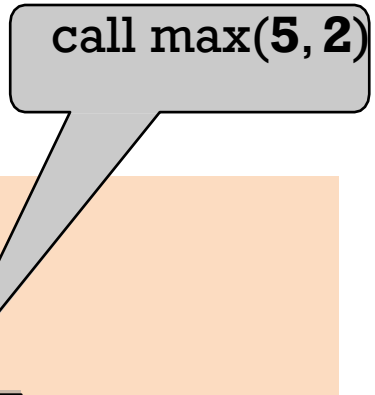
Flow of Execution (continued)




```
void main()
{
  int i = 5;
  int j = 2;
  int k = max( i, j );

cout << "The maximum between " << i <<
   "and " << j << " is " << k;

}
```

```
int max(int num1, int num2}
{
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

Flow of Execution (continued)




```
void main()
{
  int i = 5;
  int j = 2;
  int k = max( i, j );

cout << "The maximum between " << i <<
   "and " << j << " is " << k;

}
```

```
int max(int num1, int num2}
{
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

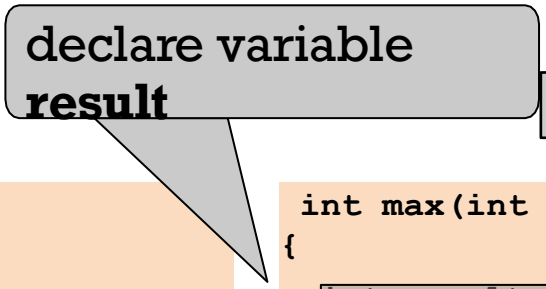
Flow of Execution (continued)

call max(**5**, **2**)
Pass the value of **5** to **num1**
Pass the value of **2** to **num2**

```
void main()
{
  int i = 5;
  int j = 2;
  int k = max( i, j );

cout << "The maximum between " << i <<
   "and " << j << " is " << k;

}
```

```
, int num2)
{
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

Flow of Execution (continued)



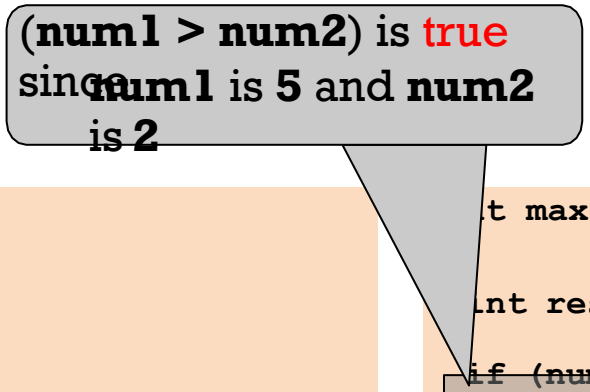
5

2

```
void main()
{
  int i = 5;
  int j = 2;
  int k = max( i, j );

cout << "The maximum between " << i <<
   "and " << j << " is " << k;

}
```

```
int max(int num1, int num2}
{
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

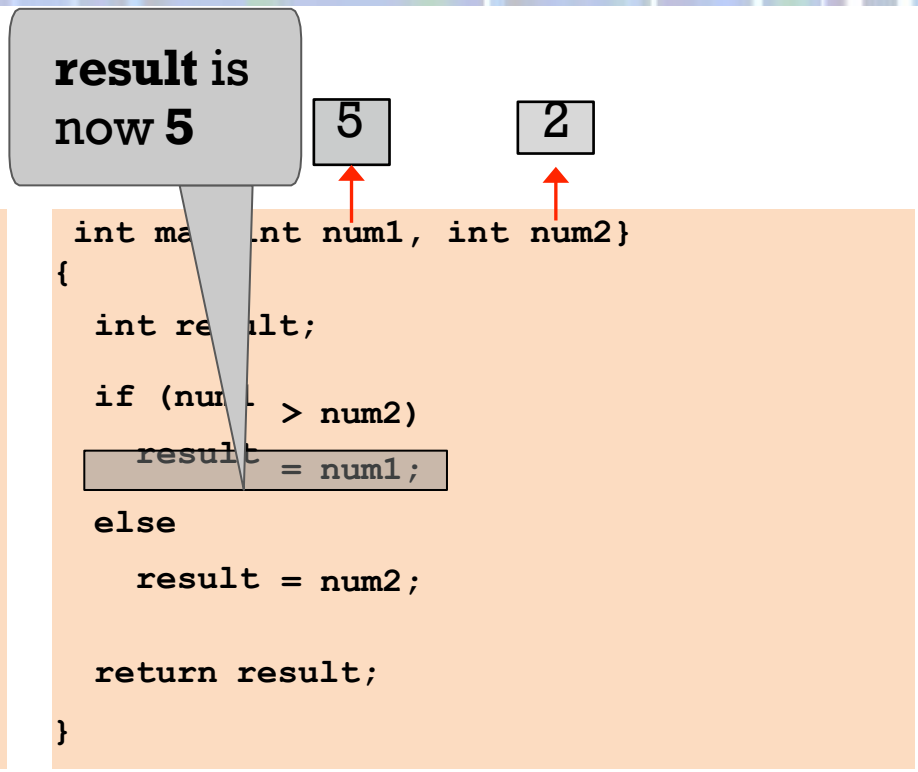
Flow of Execution (continued)




```
void main()
{
  int i = 5;
  int j = 2;
  int k = max( i, j );

cout << "The maximum between " << i <<
   "and " << j << " is " << k;

}
```

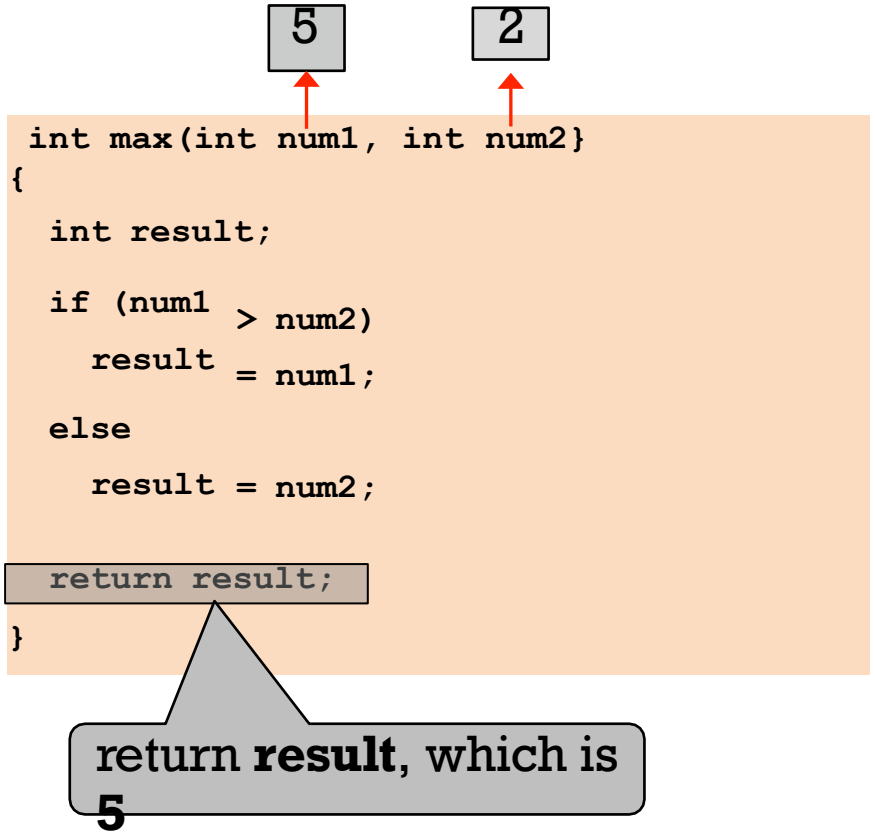
```
int max(int num1, int num2)

  int result;

  if (num1 > num2)
     result = num1;
  else
     result = num2;

  return result;
}
```

Flow of Execution (continued)



Flow of Execution (continued)

```
void main()
{
  int i = 5;
  int j = 2;
  int k = max( i, j );

cout << "The maximum between " << i <<
   "and " << j << " is " << k;
}
```



Flow of Execution (continued)

return max(5, 2) and assign the return value to **k**

```
void main()
{
  int i = 5;
  int j = 2;
  int k = max( i, j );

cout << "The maximum between " << i <<
   "and " << j << " is " << k;

}
```

```
int max(int num1, int num2}
{
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

Flow of Execution (continued)

Execute the **cout** statement

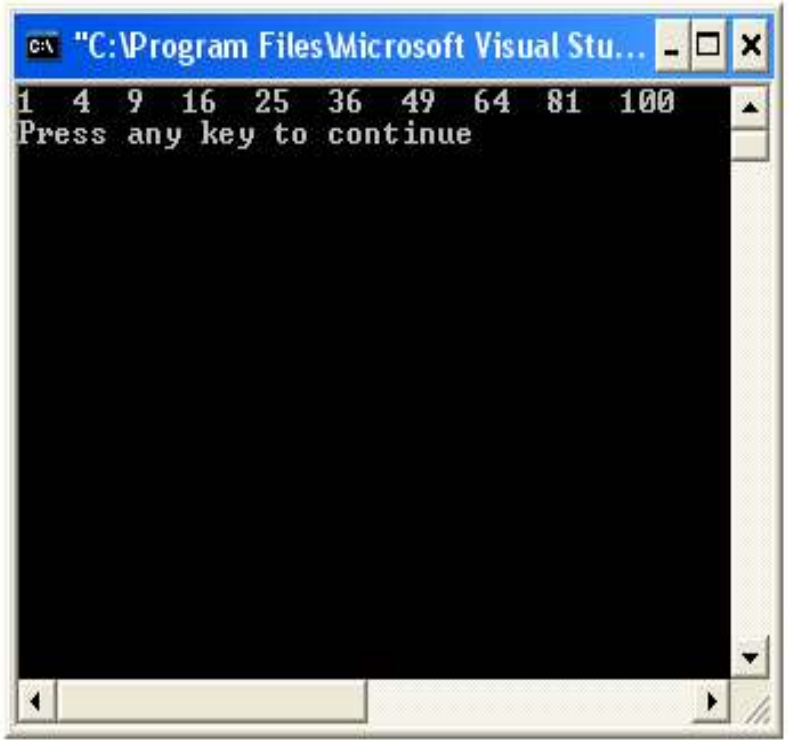
```
void main()
{
  int i = 5;
  int j = 2;
  int k = max( i, j );

cout << "The maximum between " << i <<
   "and " << j << " is " << k;
}
```

```
 int max(int num1, int num2}
{
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

Programmer-Defined Functions (Example)

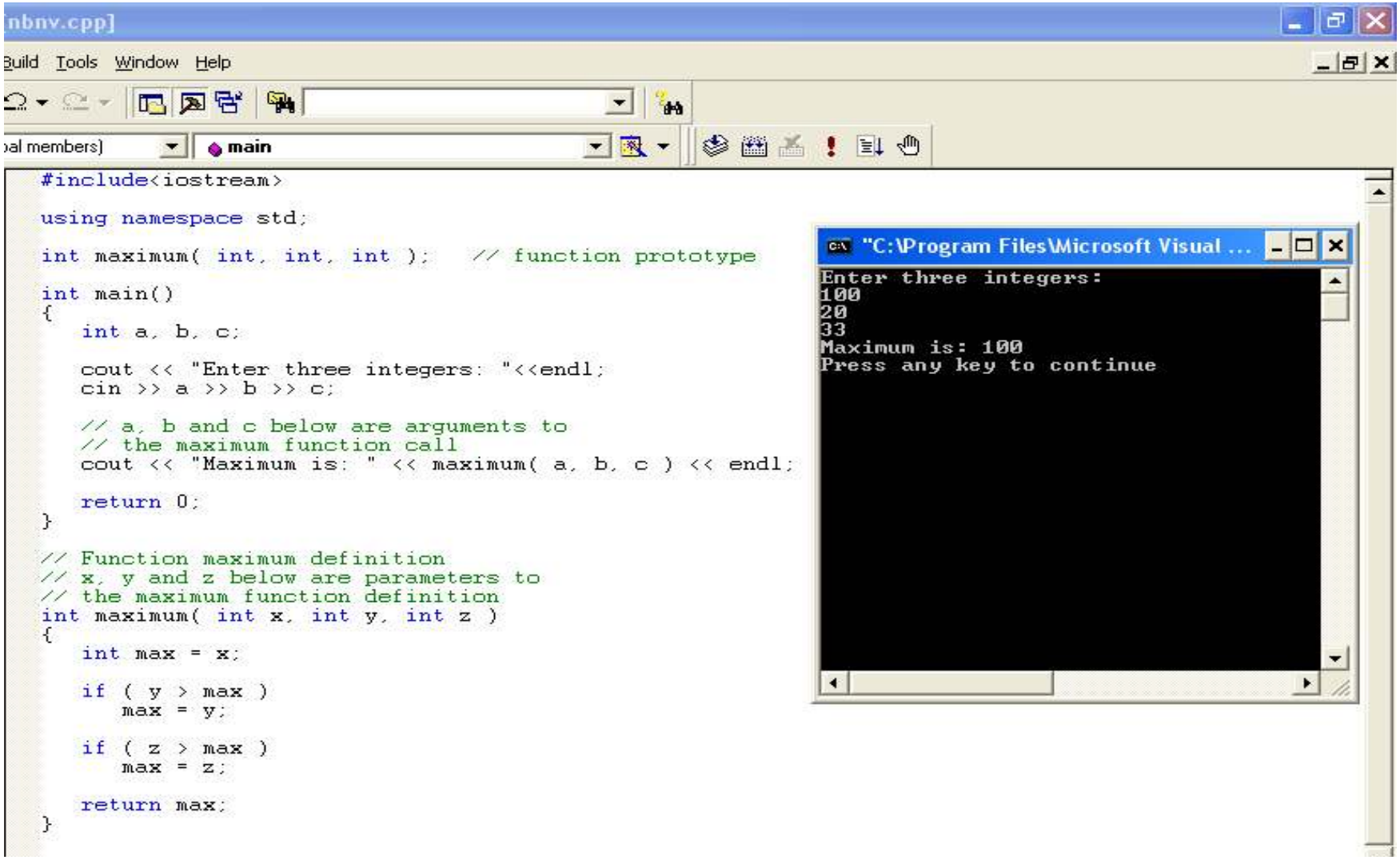
```
#include<iostream>

using namespace std;
int square( int );   // function prototype
int main()
{
   for ( int x = 1; x <= 10; x++ )
      cout << square( x ) << "  ";

   cout << endl;
   return 0;
}
// Function definition
int square( int y )
{
   return y * y;
}
```

```cpp
#include<iostream>

using namespace std;
int square( int );   // function prototype

int main()
{
   for ( int x = 1; x <= 10; x++ )
      cout << square( x ) << "   ";

   cout << endl;
   return 0;
}

// Function definition
int square( int y )
{
   return y * y;
}
```



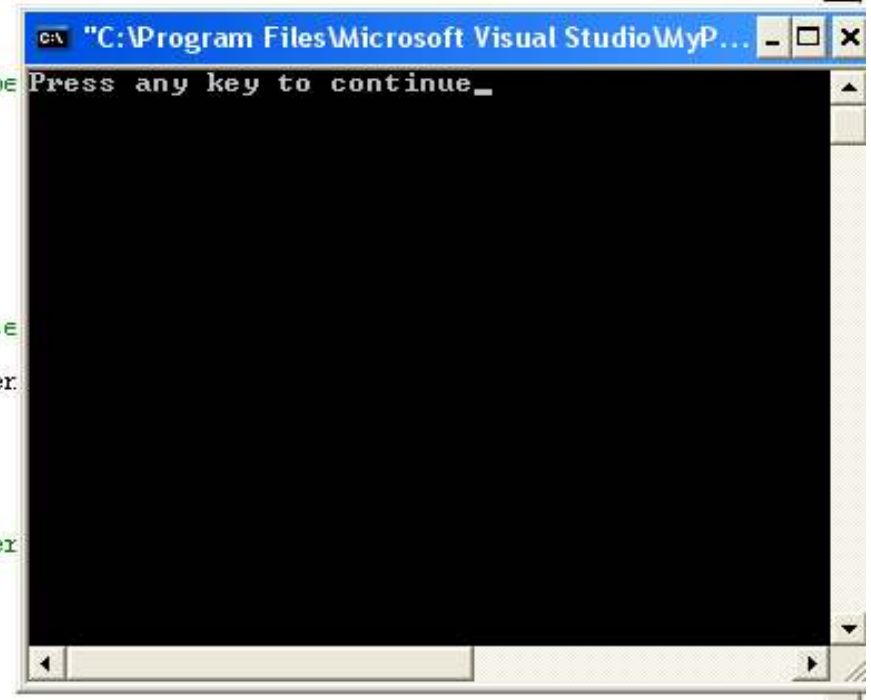
Programmer-Defined Functions (Example)

```
#include<iostream>
using namespace std;
int maximum( int, int, int );   // function prototype
int main()
{
   int a, b, c;

   cout << "Enter three integers: "<<endl;
   cin >> a >> b >> c;

   // a, b and c below are arguments (actual parameters) to
   // the maximum function call
   cout << "Maximum is: " << maximum( a, b, c ) << endl;

   return 0;
}
// Function maximum definition
// x, y and z below are parameters ( formal parameters) to
// the maximum function definition
int maximum( int x, int y, int z )
{
   int max = x;

   if ( y > max )
      max = y;
   if ( z > max )
      max = z;
   return max;
}
```


nbnv.cpp]
Build Tools Window Help
al members)
main
#include<iostream>
using namespace std;
int maximum(int, int, int); // function prototype
int main()
{
int a, b, c;
cout << "Enter three integers: "<<endl;
cin >> a >> b >> c;
// a, b and c below are arguments to
// the maximum function call
cout << "Maximum is: " << maximum(a, b, c) << endl;
return 0;
}
// Function maximum definition
// x, y and z below are parameters to
// the maximum function definition
int maximum(int x, int y, int z)
{
int max = x;
if (y > max)
max = y;
if (z > max)
max = z;
return max;
}
"C:\Program Files\Microsoft Visual ...
Enter three integers:
100
20
33
Maximum is: 100
Press any key to continue

Why is this program not doing anything?

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

int maximum( int, int, int );   // function prototype

int main()
{
   int a, b, c;
 return 0;
   cout << "Enter three integers: "<<endl;
   cin >> a >> b >> c;

   // a, b and c below are arguments (actual paramete
   // the maximum function call
   cout << "Maximum is: " << maximum( a, b, c ) << en

   return 0;
}

// Function maximum definition
// x, y and z below are parameters ( formal parameter
// the maximum function definition
int maximum( int x, int y, int z )
{
   int max = x;

   if ( y > max )
      max = y;
   if ( z > max )
      max = z;
   return max;
}
```



void-Functions

- In top-down design, a subtask might produce
  - No value (just input or output for example)
  - One value
  - More than one value (using pointers)
- We have seen how to implement functions that return one value
- A void-function implements a subtask that returns no value or more than one value

void-Function Definition

- Two main differences between void-function definitions and the definitions of functions that return one value
  - Keyword void replaces the type of the value returned
    - void means that no value is returned by the function
  - The return statement does not include and expression
- Example:

```
void showResults(double f_degrees, double c_degrees)
 {

      cout << f_degrees
           <<“  degrees Fahrenheit is euivalent to” << endl
           << c_degrees << “  degrees Celsius.” << endl;
       return;
}
```

Using a void-Function

- void-function calls are executable statements
  - They do not need to be part of another statement
  - They end with a semi-colon
- Example:

```
showResults(32.5, 0.3);
```

NOT: `cout << showResults(32.5, 0.3);`

void-Function Calls

- Mechanism is nearly the same as the function calls we have seen
  - Argument values are substituted for the formal parameters
    - It is fairly common to have no parameters in void-functions
      - In this case there will be no arguments in the function call
  - Statements in function body are executed
  - Optional return statement ends the function
    - Return statement does not include a value to return
    - Return statement is implicit if it is not included

void-Functions
Why Use a Return?

- Is a return-statement ever needed in a void-function since no value is returned?
  - Yes!
    - What if a branch of an if-else statement requires that the function ends to avoid producing more output, or creating a mathematical error?
    - void-function in Display 5.3, avoids division by zero with a return statement

Display 5.3

DISPLAY 5.3 Use of *return* in a *void* Function

Function Declaration

```
1    void iceCreamDivision(int number, double totalWeight);
2    //Outputs instructions for dividing totalWeight ounces of
3    //ice cream among number customers.
4    //If number is 0, nothing is done.
```

Function Definition

```
1    //Definition uses iostream:
2    void iceCreamDivision(int number, double totalWeight)
3    {
4        using namespace std;
5        double portion;
6
7        if (number == 0)
8            return;
9        portion = totalWeight/Number;
10       cout.setf(ios::fixed);
11       cout.setf(ios::showpoint);
12       cout.precision(2);
13       cout << "Each one receives "
14            << portion << " ounces of ice cream." << endl;
15   }
```

If number is 0, then the function execution ends here.

Creating Program-Defined Void Functions

```
 1  //ABC.cpp - displays the total sales
 2  //Created/revised by <your name> on <current date>
 3
 4  #include <iostream>
 5  using namespace std;
 6
 7  //function prototypes
 8  void displayLine();
 9  void displayCompanyInfo();
10  void displayTotalSales(int total);
11
12  int main()
13  {
14      int store1Sales = 0;
15      int store2Sales = 0;
16      int totalSales  = 0;
17
18      //enter input items
19      cout << "Store 1's sales: ";
20      cin >> store1Sales;
21      cout << "Store 2's sales: ";
22      cin >> store2Sales;
23
24      //calculate total sales
25      totalSales = store1Sales + store2Sales;
26
```

function prototypes end with a semicolon

Figure 10-5 ABC Company program

Creating Program-Defined Void Functions (cont'd.)

```
27      //display output items
28      displayLine();
29      displayCompanyInfo();
30      displayTotalSales(totalSales);
31      displayLine();
32
33      system("pause");
34      return 0;
35  }   //end of main function
36
37  //*****function definitions*****
38  void displayLine()
39  {
40      cout << "--------------------" << endl;
41  }   //end of displayLine function
42
43  void displayCompanyInfo()
44  {
45      cout << "ABC Company" << endl;
46      cout << "Chicago, Illinois" << endl << endl;
47  }   //end of displayCompanyInfo function
48
49  void displayTotalSales(int total)
50  {
51      cout << "Total sales: $" << total << endl;
52  }   //end of displayTotalSales function
```

your C++ development tool may not require this statement

function headers do not end with a semicolon

Figure 10-5 ABC Company program (cont'd.)

DISPLAY 4.3 A Function Definition

```
1   #include <iostream>
2   using namespace std;
3
4   double totalCost(int numberPar, double pricePar);
5   //Computes the total cost, including 5% sales tax,
6   //on numberPar items at a cost of pricePar each.
7
8   int main( )
9   {
10      double price, bill;
11      int number;
12
13      cout << "Enter the number of items purchased: ";
14      cin >> number;
15      cout << "Enter the price per item $";
16      cin >> price;
17
18      bill = totalCost(number, price);
19
20      cout.setf(ios::fixed);
21      cout.setf(ios::showpoint);
22      cout.precision(2);
23      cout << number << " items at "
24           << "$" << price << " each.\n"
25           << "Final bill, including tax, is $" << bill
26           << endl;
27
28      return 0;
29  }
30
31  double totalCost(int numberPar, double pricePar)
32  {
33      const double TAX_RATE = 0.05;  //5% sales tax
34      double subtotal;
35
36      subtotal = pricePar * numberPar;
37      return (subtotal + subtotal * TAX_RATE);
38  }
```

function declaration/function prototype

function call

function heading

function body

function definition

Function Call Details

- The values of the arguments are plugged into the formal parameters (Call-by-value mechanism with call-by-value parameters)
  - The first argument is used for the first formal parameter, the second argument for the second formal parameter, and so forth.
  - The value plugged into the formal parameter is used in all instances of the formal parameter in the function body

DISPLAY 4.4 Details of a Function Call

```
int main()
{
    double price, bill;
    int number;
    cout << "Enter the number of items purchased: ";
    cin >> number;
    cout << "Enter the price per item $";
    cin >> price;

    bill = totalCost (number, price);
                       2       10.10
    cout.setf (ios::fixed);
    cout.setf (ios::showpoint);
    cout.precision(2);
    cout << number << " items at "
         << "$" << price << " each.\n"
21.21    << "Final bill, including tax, is $" << bill
         << endl;
    return 0;
}
                                  2          10.10
double totalCost (int numberPar, double pricePar)
{
    const double TAX_RATE = 0.05; //5% sales tax
    double subtotal;

    subtotal = pricePar * numberPar;
    return (subtotal + subtotal * TAX_RATE);
}
```

1. Before the function is called, values of the variables number and price are set to 2 and 10.10, by cin statements (as you can see the Sample Dialogue in Display 4.3)

2. The function call executes and the value of `number` (which is 2) plugged in for `numberPar` and value of `price` (which is 10.10) plugged in for `pricePar`.

3. The body of the function executes with `numberPar` set to 2 and `pricePar` set to10.10, producing the value 20.20 in subtotal.

4. When the return statement is executed, the value of the expression after return is evaluated and returned by the function. In this case, `(subtotal + subtotal * TAX_RATE)` is (20.20 + 20.20*0.05) or 21.21.

5. The value 21.21 is returned to where the function was invoked. The result is that `totalCost (number, price)` is replaced by the return value of 21.21. The value of `bill` (on the left-hand side of the equal sign) is set equal to 21.21 when the statement `bill = totalCost (number, price);` finally ends.

Alternate Declarations

- Two forms for function declarations (prototypes).
  - List formal parameter names
  - List types of formal parmeters, but not names
- Examples:

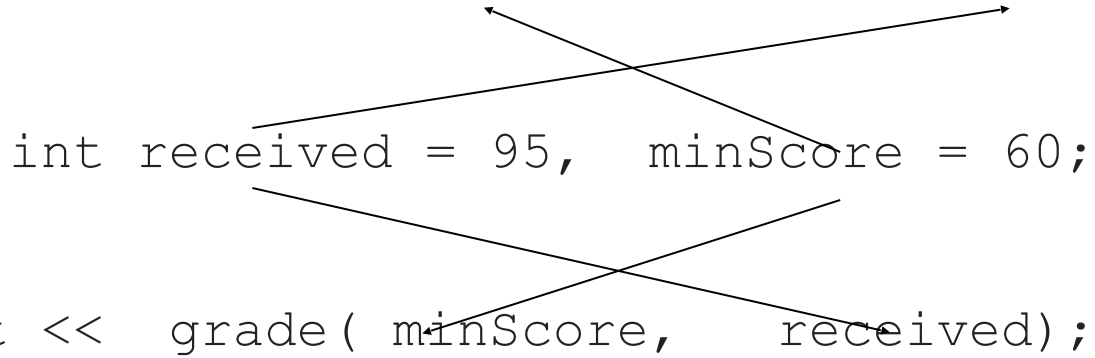
```
double totalCost(int numberPar, double pricePar);

double totalCost(int, double);
```

- First aids description of the function in comments
- Function headers must always list formal parameter names!

Order of Arguments

- Compiler checks that the types of the arguments are correct and in the correct sequence.
- Compiler cannot check that arguments are in the correct logical order
- Example: Given the function declaration:

```
char grade(int received_par,   int minScore_par);
```



```
int received = 95,  minScore = 60;

cout <<  grade( minScore,   received);
```

  - Produces a faulty result because the arguments are not in the correct logical order. The compiler will not catch this!

DISPLAY 4.5 Incorrectly Ordered Arguments

```
1    //Determines user's grade. Grades are Pass or Fail.
2    #include <iostream>
3    using namespace std;
4
5    char grade(int receivedPar, int minScorePar);
6    //Returns 'P' for passing, if receivedPar is
7    //minScorePar or higher. Otherwise returns 'F' for failing.
8
9    int main( )
10   {
11       int score, needToPass;
12       char letterGrade;
13
14       cout << "Enter your score"
15            << " and the minimum needed to pass:\n";
16       cin >> score >> needToPass;
17
18       letterGrade = grade(needToPass, score);
19
```

DISPLAY 4.5 Incorrectly Ordered Arguments

```
20          cout << "You received a score of " << score << endl
21               << "Minimum to pass is " << needToPass << endl;
22
23          if (letterGrade == 'P')
24              cout << "You Passed. Congratulations!\n";
25          else
26              cout << "Sorry. You failed.\n";
27
28          cout << letterGrade
29               << " will be entered in your record.\n";
30
31          return 0;
32      }
33
34      char grade(int receivedPar, int minScorePar)
35      {
36          if (receivedPar >= minScorePar)
37              return 'P';
38          else
39              return 'F';
40      }
```

Sample Dialogue

```
Enter your score and the minimum needed to pass:
98 60
You received a score of 98
Minimum to pass is 60
Sorry. You failed.
F will be entered in your record.
```

Function Definition Syntax

- Within a function definition
  - Variables must be declared before they are used
  - Variables are typically declared before the executable statements begin
  - At least one return statement must end the function
    - Each branch of an if-else statement might have its own return statement

bool Return Values

- A function can return a `bool` value
  - Such a function can be used where a boolean expression is expected
    - Makes programs easier to read
- `if (((rate >=10) && ( rate < 20)) || (rate == 0))` is easier to read as

```
if (appropriate (rate))
```

  - If function appropriate returns a `bool` value based on the expression above

Function appropriate

- To use function appropriate in the if-statement

```
if (appropriate (rate))
{    …    }
```

appropriate could be defined as

```
bool appropriate(int rate)
{
 return(((rate >=10) && ( rate < 20)) || (rate == 0));
}
```

User-defined Function Conclusion

- Can you
  - Write a function declaration and a function definition for a function that takes three arguments, all of type `int`, and that returns the sum of its three arguments?
  - Describe the call-by-value parameter mechanism?
  - Write a function declaration and a function definition for a function that takes one argument of type `int` and one argument of type `double`, and that returns a value of type `double` that is the average of the two arguments?

Exercise

- Write the C++ code for a function that receives an integer passed to it. The function should divide the integer by 2 and then return the result, which may contain a decimal place. Name the function `divideByTwo`. Name the formal parameter `wholeNumber`.

```
double divideByTwo(int wholeNumber)
{
      return wholeNumber / 2.0;
}
```

PASSING BY REFERENCE
PASSING BY VALUES

Passing Variables by Reference

- Passing a variable's address in internal memory to a function is referred to as passing *by reference*
- You pass *by reference* when you want the receiving function **to change the contents of the variable**

- To pass *by reference* in C++, you include an ampersand (**&**) before the name of the formal parameter in the receiving function's header
- Ampersand (**&**) is the **address-of operator**
  - Tells the computer to pass the variable's address rather than a copy of its contents

Passing Variables by Reference (cont'd.)

- If receiving function appears below `main`, you must also include the **&** in the receiving function's prototype

- You enter the **&** immediately before the name of the formal parameter in the prototype
  - If the prototype does not contain the formal parameter's name, you enter a space followed by **&** after the formal parameter's data type

- Void functions use variables passed *by reference* to send information back to the calling function, instead of a `return` value

Passing Variables by Reference (cont'd.)

```
#include<iostream>
using namespace std;
void Test(int &);
int  main()
{
    int a;
    cin>>a;
    cout<<"Before calling  "<<a<<endl;
    Test(a);
    cout<<"After calling  "<<a<<endl;
    return 0;
}

void Test(int &b)
{
    b=b+b;
}
```

```
   5
Before calling  5
After calling  10
```

Passing Variables by Reference (cont'd.)

```
1  //Modified Age.cpp - displays the user's age in a message
2  //Created/revised by <your name> on <current date>
3
4  #include <iostream>
5  using namespace std;
6
7  //function prototypes
8  void getAge(int &inYears);
9  void displayAge(int years);
10
11 int main()
12 {
13     int age = 0;
14
15     getAge(age);
16
17     displayAge(age);
18
19     //system("pause");
20     return 0;
21 }   //end of main function
22
23 //*****function definitions*****
24 void getAge(int &inYears)
25 {
26     cout << "How old are you? ";
27     cin >> inYears;
28 }   //end of getAge function
29
30 void displayAge(int years)
31 {
32     cout << "You are " << years << " years old." << endl;
33 }   //end of displayAge function
```

address-of operator

you also can use void getAge(int &);

address-of operator

Figure 10-13 Modified age message program

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
```

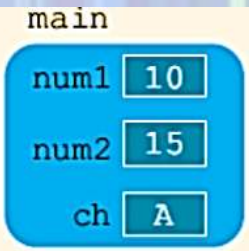


FIGURE 7-5 Values of the variables after the statement in Line 3 executes

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

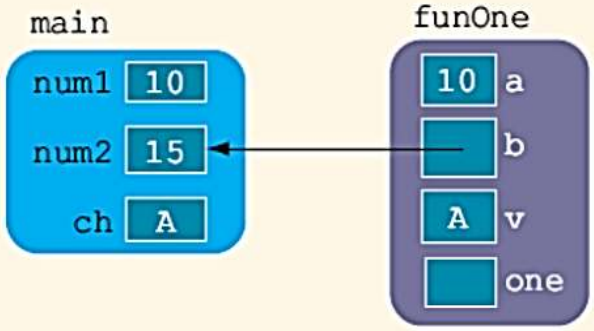


FIGURE 7-6 Values of the variables just before the statement in Line 9 executes

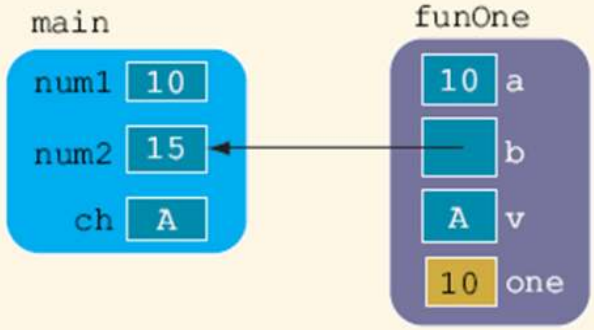


FIGURE 7-7 Values of the variables after the statement in Line 9 executes

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

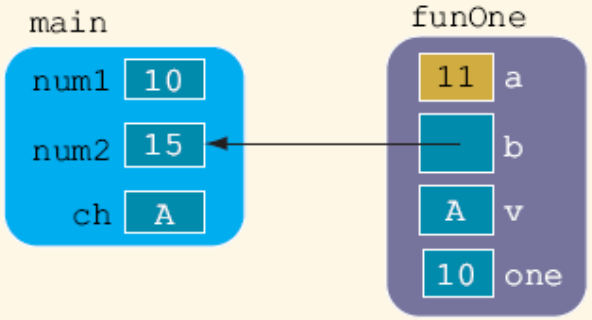


FIGURE 7-8 Values of the variables after the statement in Line 10 executes

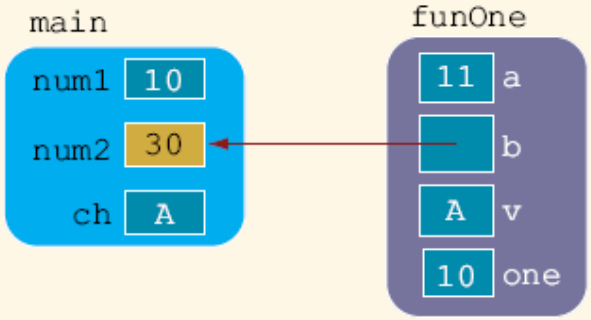


FIGURE 7-9 Values of the variables after the statement in Line 11 executes

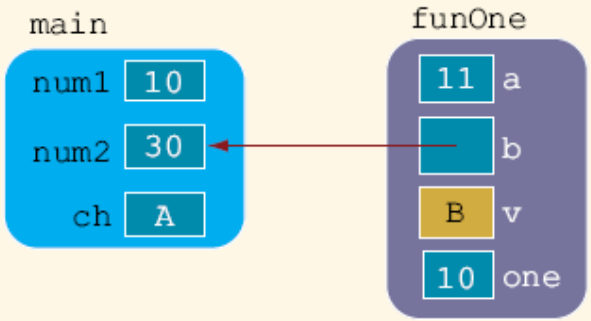


FIGURE 7-10 Values of the variables after the statement in Line 12 executes

```
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
```

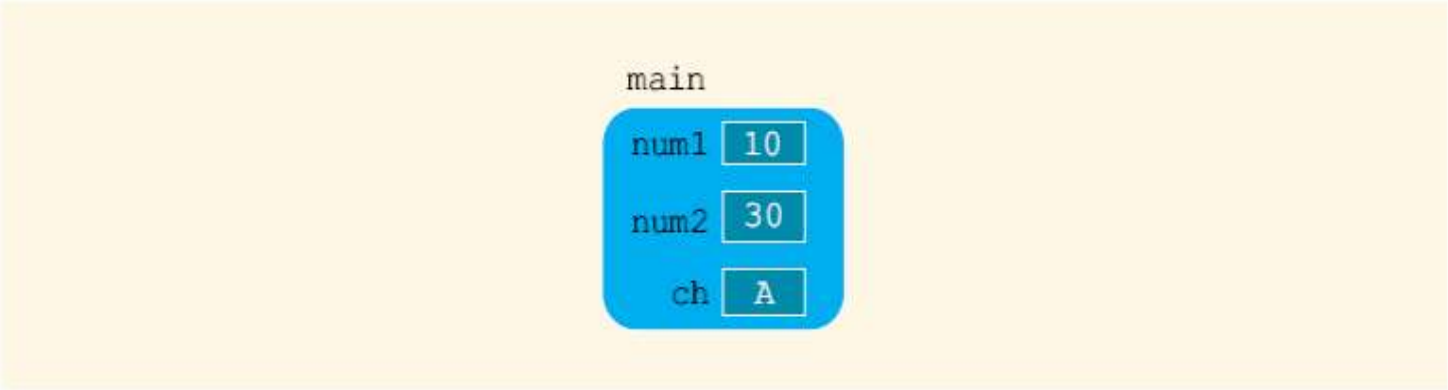


FIGURE 7-11 Values of the variables when control goes back to Line 6

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
```

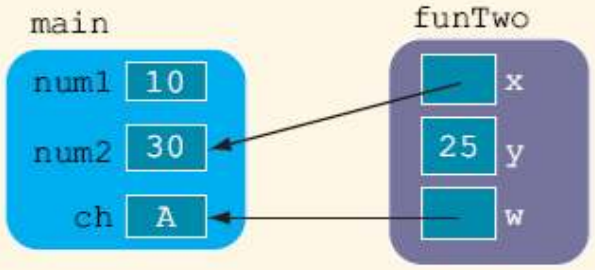


FIGURE 7-12 Values of the variables before the statement in Line 14 executes

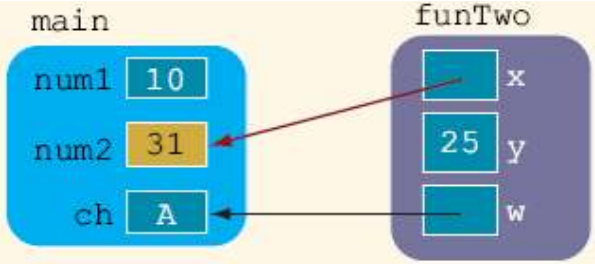


FIGURE 7-13 Values of the variables after the statement in Line 14 executes

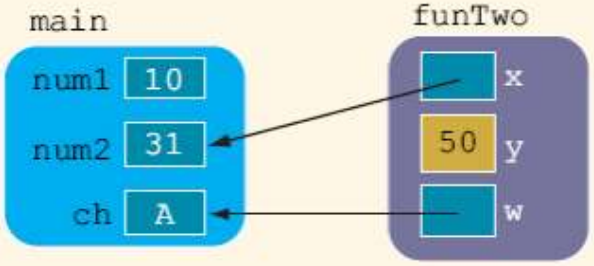


FIGURE 7-14 Values of the variables after the statement in Line 15 executes

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

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

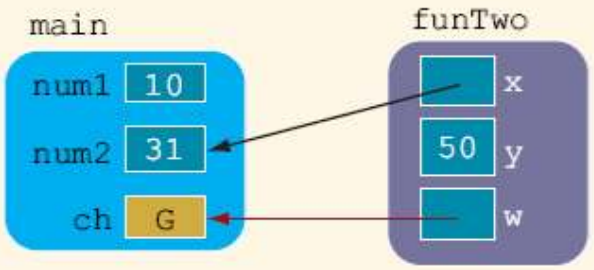


FIGURE 7-15 Values of the variables after the statement in Line 16 executes

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
```

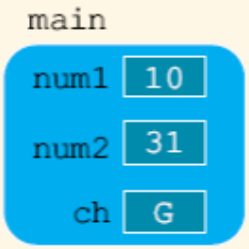


FIGURE 7-16 Values of the variables when control goes to Line 8

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

SCOPE OF VARIABLES

The Scope and Lifetime of a Variable

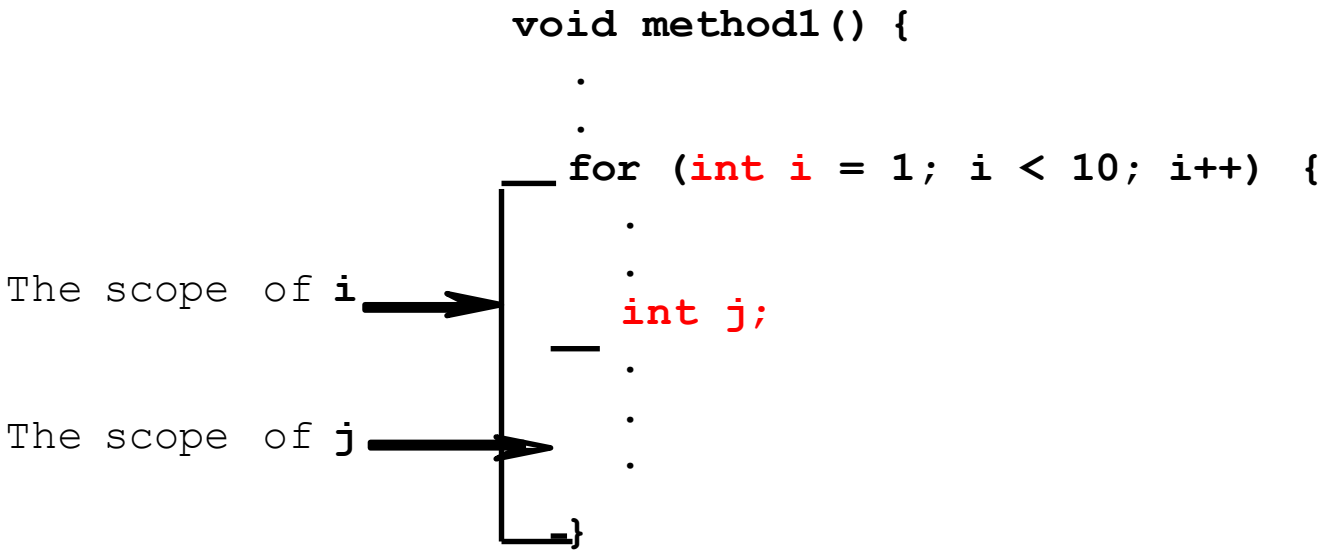
- A variable's **scope** indicates where in the program the variable can be used
- A variable's **lifetime** indicates how long the variable remains in the computer's internal memory
- Both scope and lifetime are **determined by where you declare the variable in the program**
  - Variables declared within a function and those that appear in a function's *parameterList* have a local scope and are referred to as local variables

The Scope and Lifetime of a Variable (cont'd.)

- **Local variables** can be used only by the function in which they are declared or in whose *parameterList* they appear
  - Remain in internal memory **until the function ends**
  - The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable. A local variable must be declared before it can be used
  - You can declare a local variable with the same name multiple times in different non-nesting blocks in a function, but avoid declaring local variables twice in nested blocks
  - If two methods each have a local variable of the same name, they are still two entirely different variables

The Scope and Lifetime of a Variable (cont'd.)

A variable declared in the initial action part of a **for** loop header has its scope in the entire loop. But a variable declared inside a for loop body has its scope limited in the loop body from its declaration and to the end of the block that contains the variable



The Scope and Lifetime of a Variable (cont'd.)

It is **OK** to declare **i** in two non-nested blocks

```
void method1() {
  int x = 1;
  int y = 1;

  for (int i = 1; i < 10; i++) {
    x += i;
  }

  for (int i = 1; i < 10; i++) {
    y += i;
  }
}
```

Avoid declaring same variable name in nested blocks.
Note the different **i** variables **with different scopes**

```
void method2() {

  int i = 1;
  int sum = 0;

  for (int i = 1; i < 10; i++)
    sum += i;
  }

}
```

It is **WRONG** to redefine **y**. There is different **x** variables with different scopes

```
void incorrectMethod(int x, int y) {
  int y = 1, sum = 0;

  for (int x = 1; x < 10; x++) {
    sum += x;
  }
}
```

The Scope and Lifetime of a Variable (cont'd.)

- **Global variables** are declared outside of any function in the program
  - Remain in memory **until the program ends**

- Any statement can use a global variable
  - Example: Swap function

The Scope and Lifetime of a Variable (cont'd.)

- Declaring a variable as global can allow unintentional errors to occur
  - e.g., a function that should not have access to the variable inadvertently changes the variable's contents
- You should avoid using global variables unless necessary
- If more than one function needs to access the same variable, it is better to create a local variable in one function and pass it to other functions that need it

Example: global

```
#include <iostream>
using namespace std;
 // Global variable declaration:
int g;
 int main () {
   // Local variable declaration:
   int a, b;
    // actual initialization
   a = 10;
   b = 20;
   g = a + b;
    cout << g;
    return 0;
}
```

Output: 30

Example: same name for local and global variables

```
#include <iostream>
using namespace std;
 // Global variable declaration:
int g = 20;
 int main () {
   // Local variable declaration:
   int g = 10;
    cout << g;
    return 0;
}
```

Output: 10

Exercise

- Write the C++ code for a function that receives four double numbers.
  - The function should calculate the average of the four numbers and then return the result. Name the function `calcAverage`.
  - Name the formal parameters `num1, num2, num3, and num4.`
  - Also write an appropriate function prototype for the `calcAverage` function.
  - In addition, write a statement that invokes the `calcAverage` function and assigns its return value to a double variable named `quotient`.
  - Use the following numbers as the actual arguments: `45.67, 8.35, 125.78, and 99.56.`

Exercise

- **Body**

```
double calcAverage(double num1, double num2, double num3,
double num4)
   {
         return (num1 + num2 + num3 + num4) / 4;
   }   //end of calcAverage function
```

- **Prototype**

```
double calcAverage(double num1, double num2, double num3,
double num4);
```

OR: `double calcAverage(double, double, double, double);`

- **Assigning Value**

```
quotient = calcAverage(45.67, 8.35, 125.78, 99.56);
```

Exercise

- Write the C++ code for a void function that receives three double variables
  - the first two *by value* and the last one *by reference*.
  - Name the formal parameters `n1, n2`, and `answer`.
  - The function should divide the `n1` variable by the `n2` variable and then store the result in the `answer` variable. Name the function `calcQuotient`.
  - Also write an appropriate function prototype for the `calcQuotient` function.
  - In addition, write a statement that invokes the `calcQuotient` function, passing it the `num1`, `num2,` and `quotient` variables.

Exercise

```
void calcQuotient(double n1, double n2, double
&answer)
   {
     answer = n1 / n2;
   } //end of calcQuotient function
```

Prototype

```
void calcQuotient(double, double, double &);
```
OR:
```
void calcQuotient(double n1, double n2, double
&answer);
```

Calling

```
 calcQuotient(num1, num2, quotient);
```

Any Questions

- Thanks for Listening ☺️