Chapter 6: Functions

Sections 6.1)6.13

Textbooks: Y. Daniel Liang, Introduction to Programming with C++, 3rd Edition

These slides were adapted by Prof. Gheith Abandah from the Computer Engineering Department of the University of Jordan for the Course: Computer Skills for Engineers (0907101)
Updated by Dr. Ashraf Suyyagh (Spring 2021)

Outline

- Introduction
- Defining a Function
- Calling a Function
- **`void`** Functions
- Passing Arguments by Value
- Modularizing Code
- Overloading Functions
- Function Prototypes
- Default Arguments
- Inline Functions
- Local, Global, and Static Local Variables
- Passing Arguments by Reference
- Constant Reference Parameters

Introduction

Find the sum of integers from **1** to **10**, from **20** to **37**, and from **35** to **49**, respectively.

Introduction

Write 3 loops

```
int sum = 0;
for (int i = 1; i <= 10; i++)
  sum += i;
cout << "Sum from 1 to 10 is " << sum << endl;

sum = 0;
for (int i = 20; i <= 37; i++)
  sum += i;
cout << "Sum from 20 to 37 is " << sum << endl;

sum = 0;
for (int i = 35; i <= 49; i++)
  sum += i;
cout << "Sum from 35 to 49 is " << sum << endl;
```

Introduction

```
int sum = 0;
for (int i = 1; i <= 10; i++)
  sum += i;
cout << "Sum from 1 to 10 is " << sum << endl;

sum = 0;
for (int i = 20; i <= 37; i++)
  sum += i;
cout << "Sum from 20 to 37 is " << sum << endl;

sum = 0;
for (int i = 35; i <= 49; i++)
  sum += i;
cout << "Sum from 35 to 49 is " << sum << endl;
```

Very similar 3 loops

Introduction

Functions can be used to define reusable code and organize and simplify code.

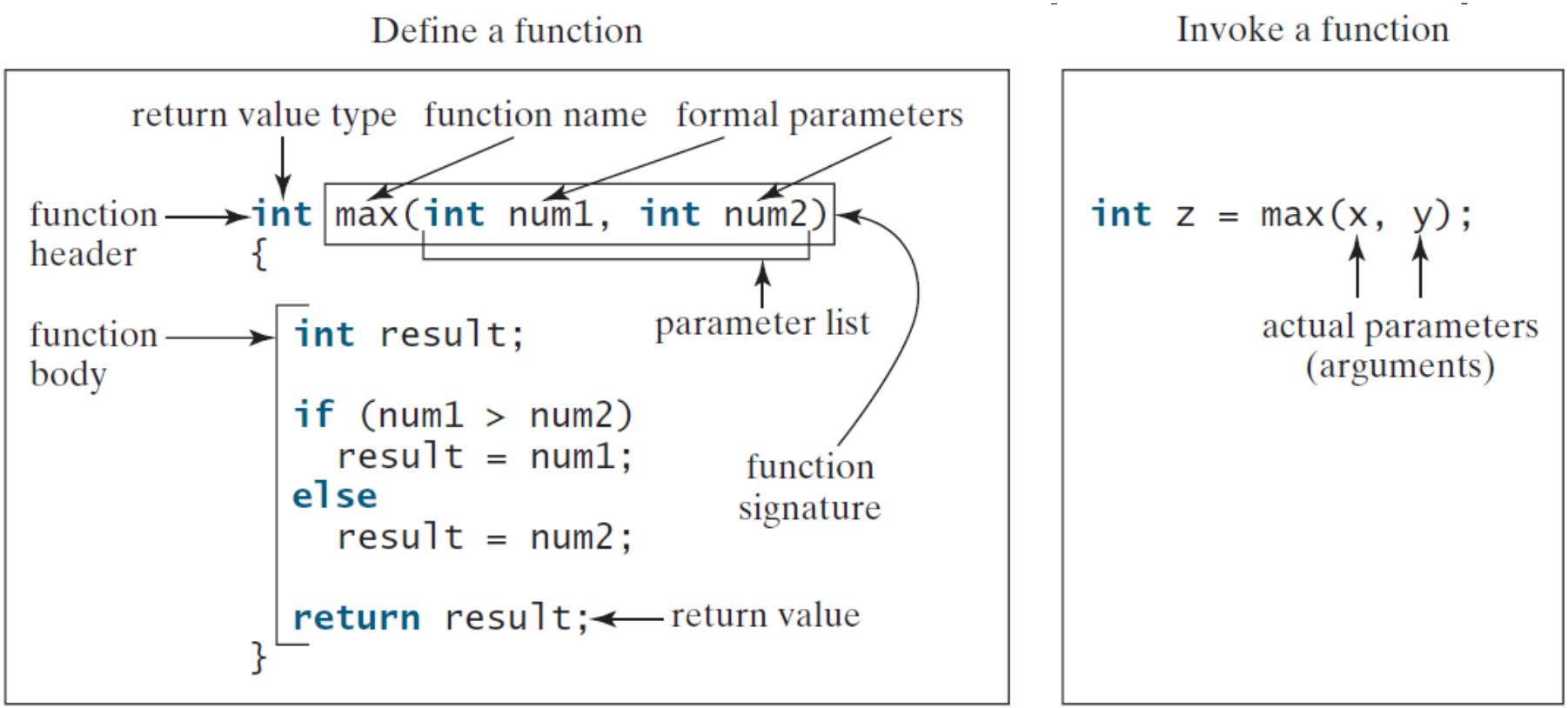
```
int sum(int i1, int i2)
{
    int sum = 0;
    for (int i = i1; i <= i2; i++)
        sum += i;
    return sum;
}

int main()
{
    cout << "Sum from 1 to 10 is " << sum(1, 10) << end
    cout << "Sum from 20 to 37 is " << sum(20, 37) <<
endl;
    cout << "Sum from 35 to 49 is " << sum(35, 49) <<
endl;
    return 0;
```

Outline

- Introduction
- Defining a Function
- Calling a Function
- **`void`** Functions
- Passing Arguments by Value
- Modularizing Code
- Overloading Functions
- Function Prototypes
- Default Arguments
- Inline Functions
- Local, Global, and Static Local Variables
- Passing Arguments by Reference
- Constant Reference Parameters

Defining a Function

- A function is a collection of statements that are grouped together to perform an operation.
- A function definition consists of its function



Defining Functions, cont.

- *Function signature* is the combination of the function name and the parameter list.
- The variables defined in the function header are known as *formal parameters*.
- When a function is invoked, you pass a value to the parameter. This value is referred to as *actual parameter* or *argument*.

Defining Functions, cont.

- A Function may return a value.
- The *return value type* is the data type of the value the function returns.
- If the function does not return a value, the return value type is the keyword *void*.

Outline

- Introduction
- Defining a Function
- Calling a Function
- **`void`** Functions
- Passing Arguments by Value
- Modularizing Code
- Overloading Functions
- Function Prototypes
- Default Arguments
- Inline Functions
- Local, Global, and Static Local Variables
- Passing Arguments by Reference
- Constant Reference Parameters

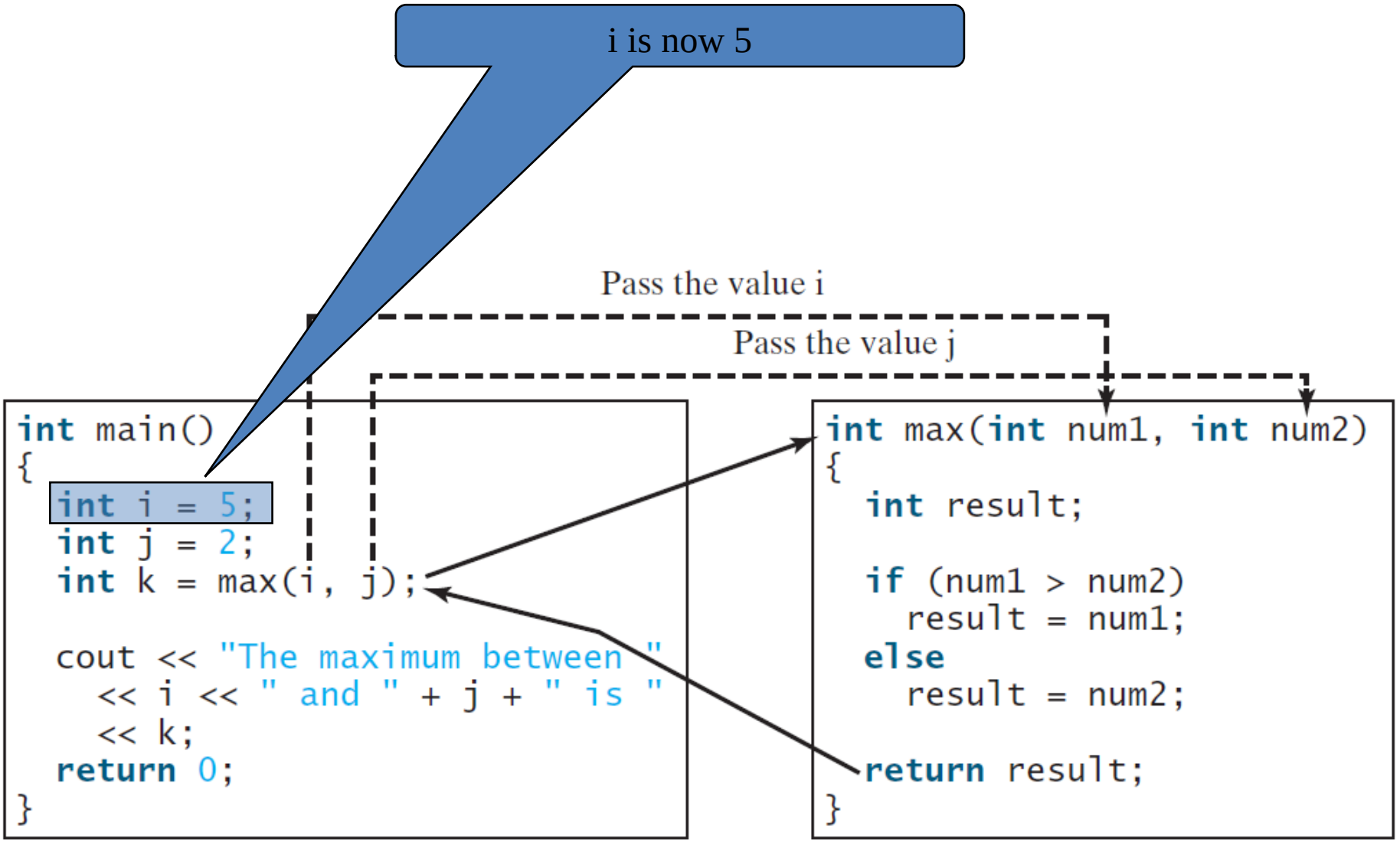
Calling a Function

This program demonstrates calling a Function max to return the largest of the **`int`** values

TestMax Run

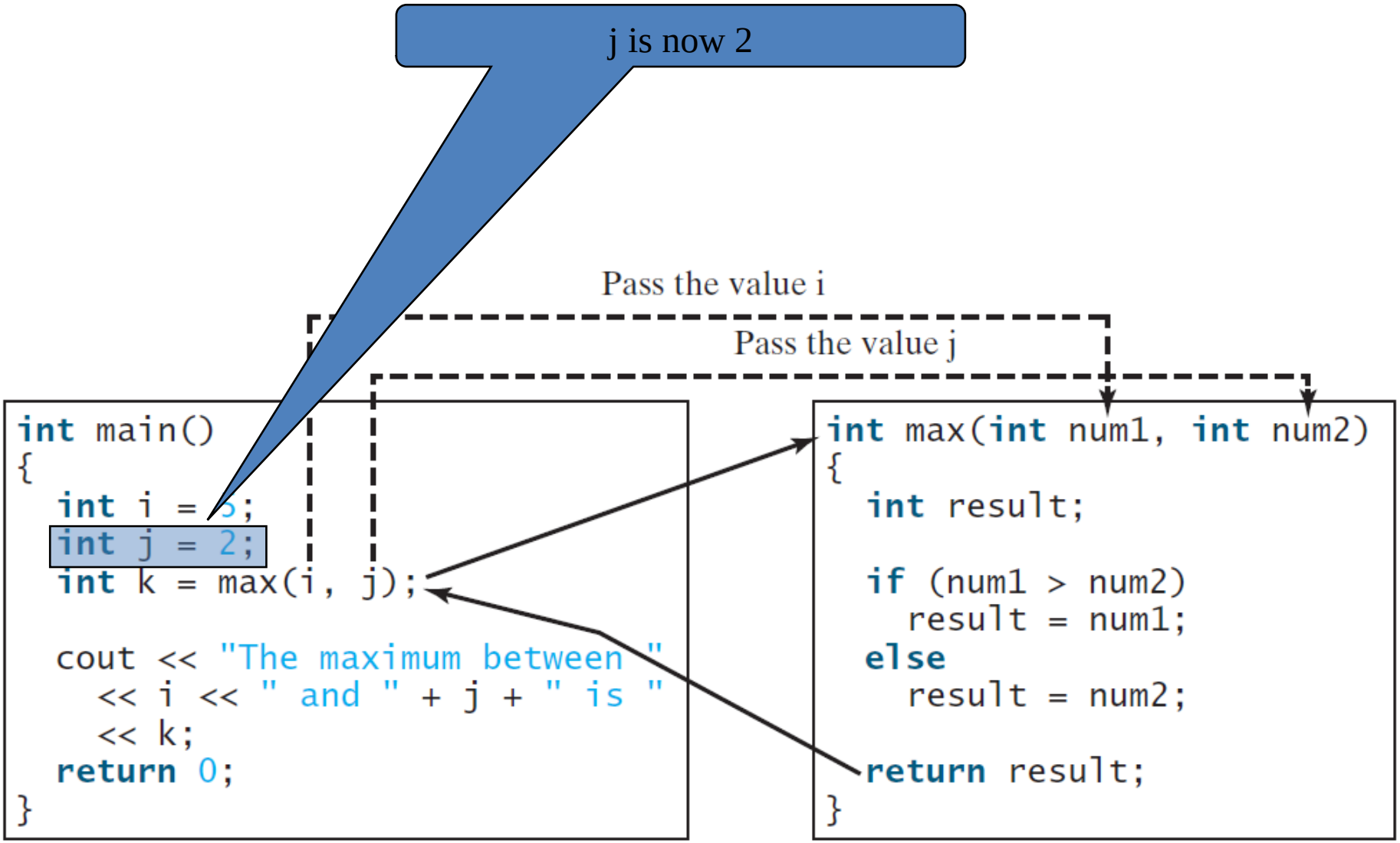
animation

Trace Function Invocation



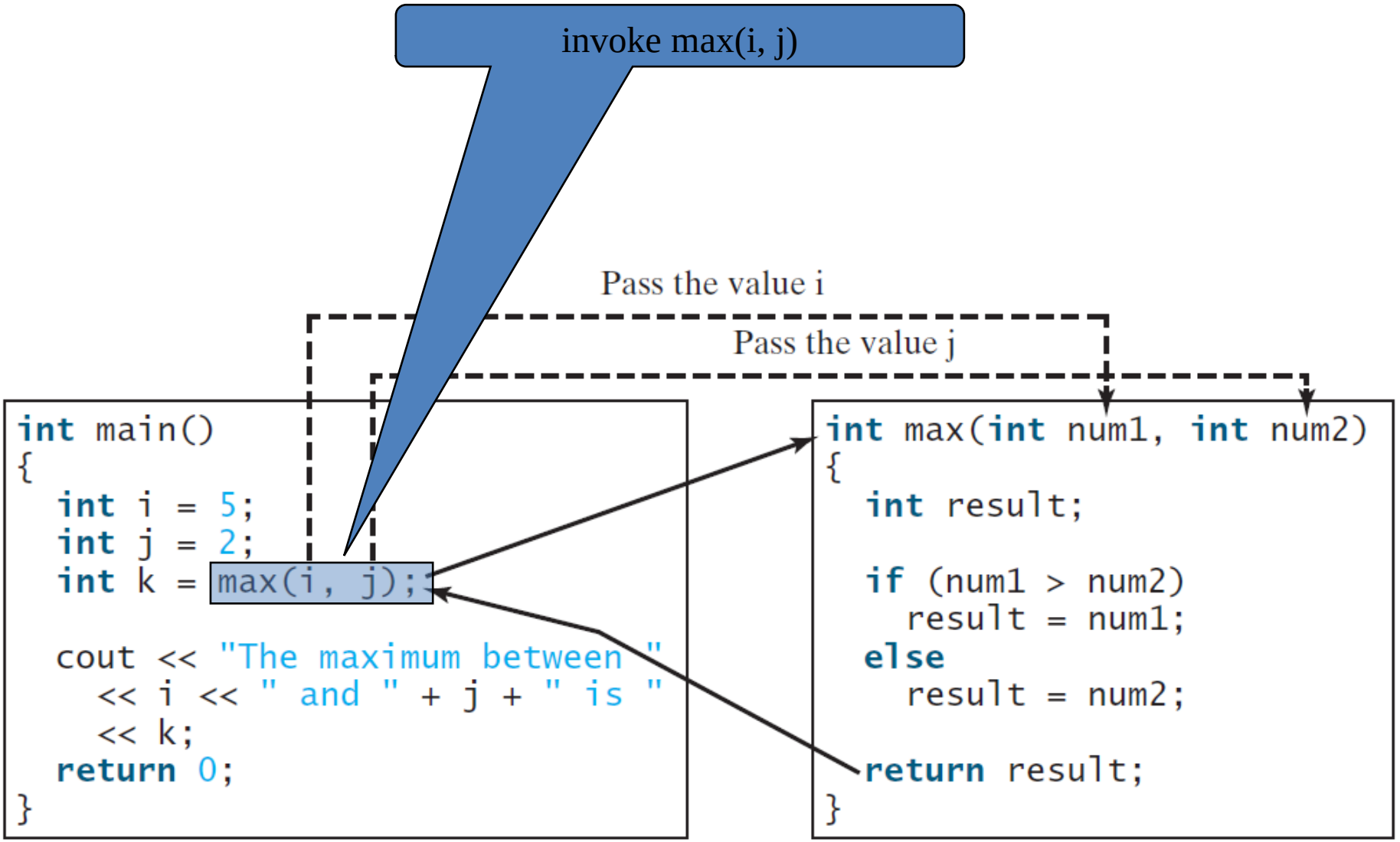
animation

Trace Function Invocation



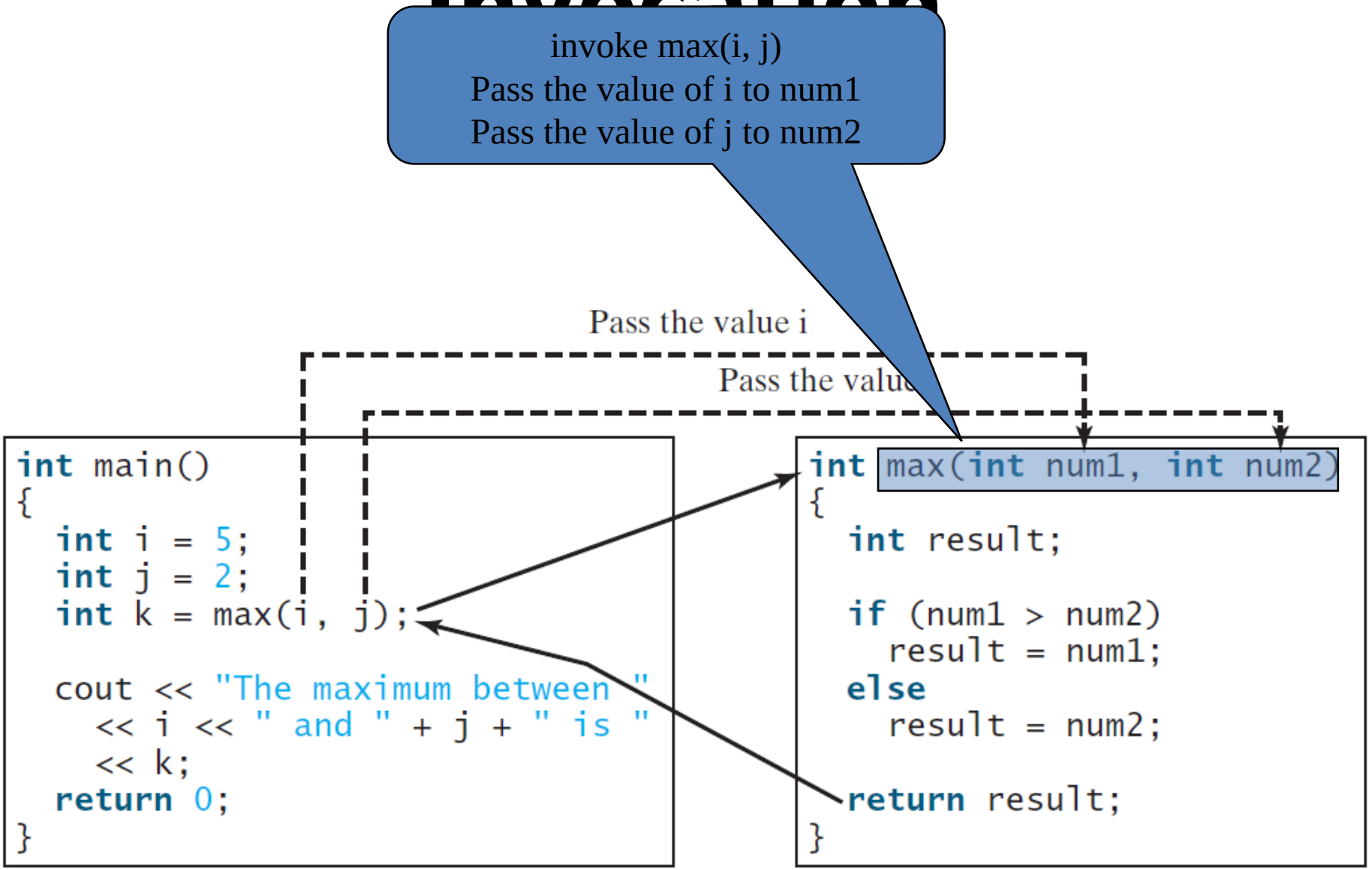
animation

Trace Function Invocation



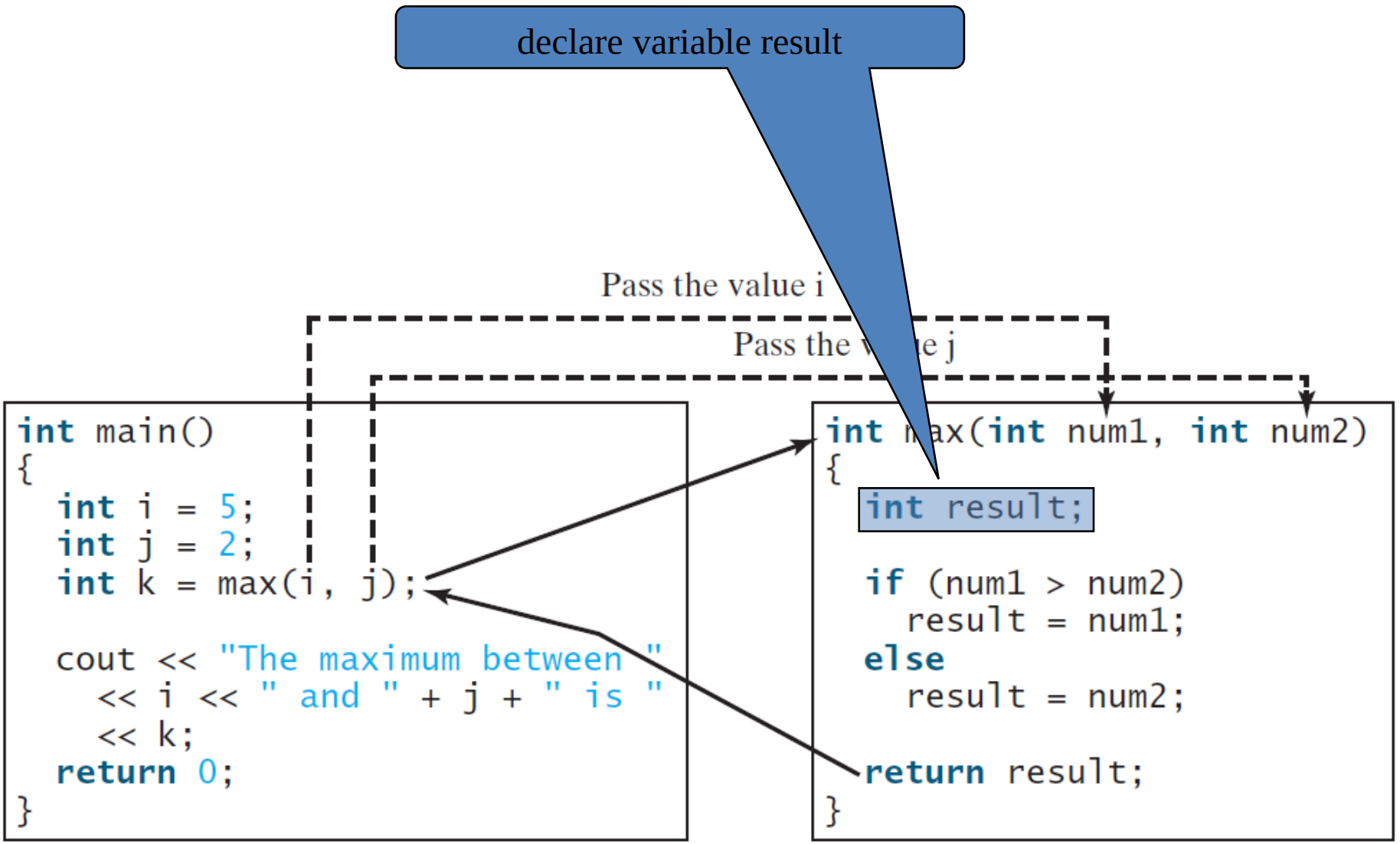
animation

Trace Function Invocation



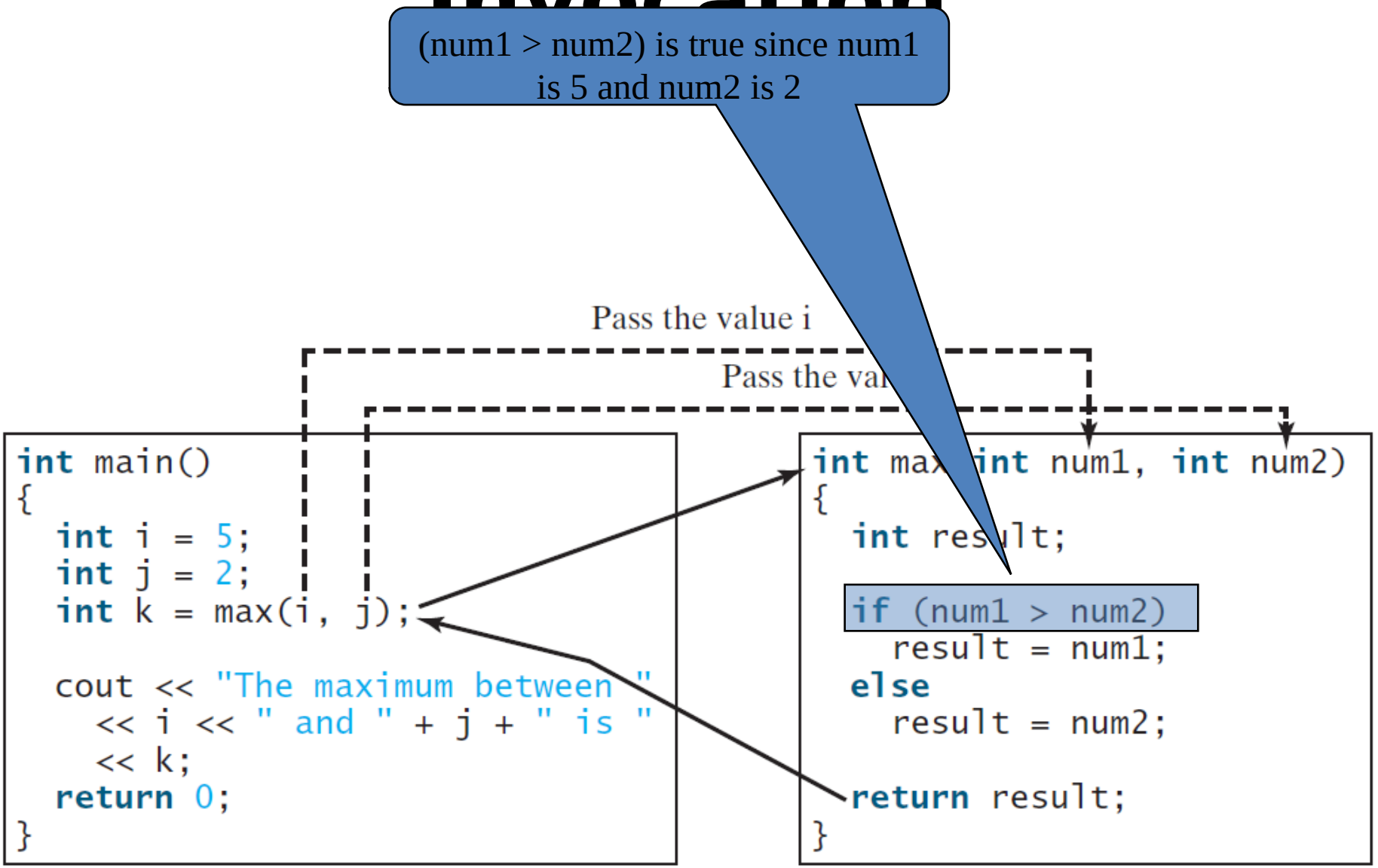
animation

Trace Function Invocation



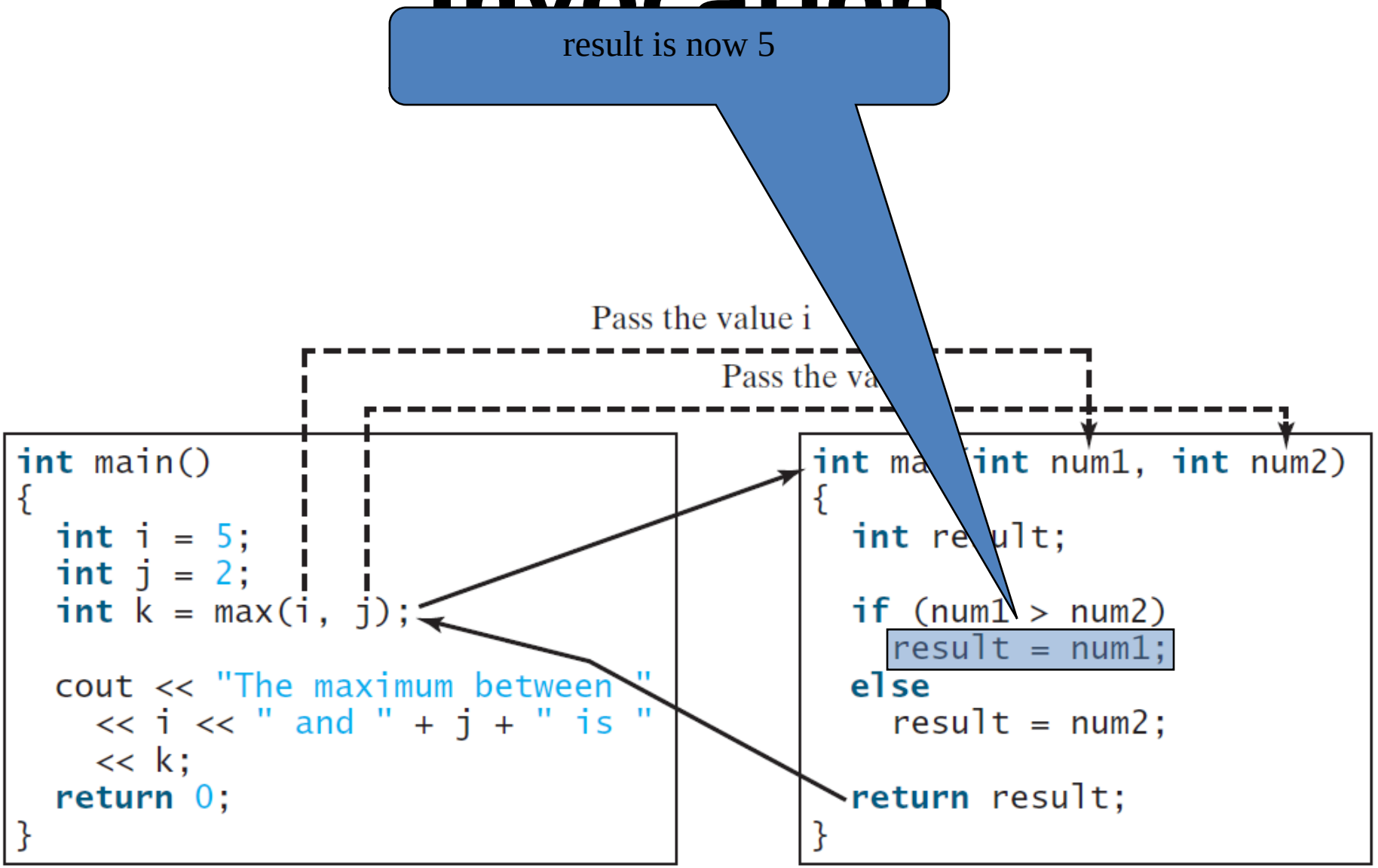
animation

Trace Function Invocation



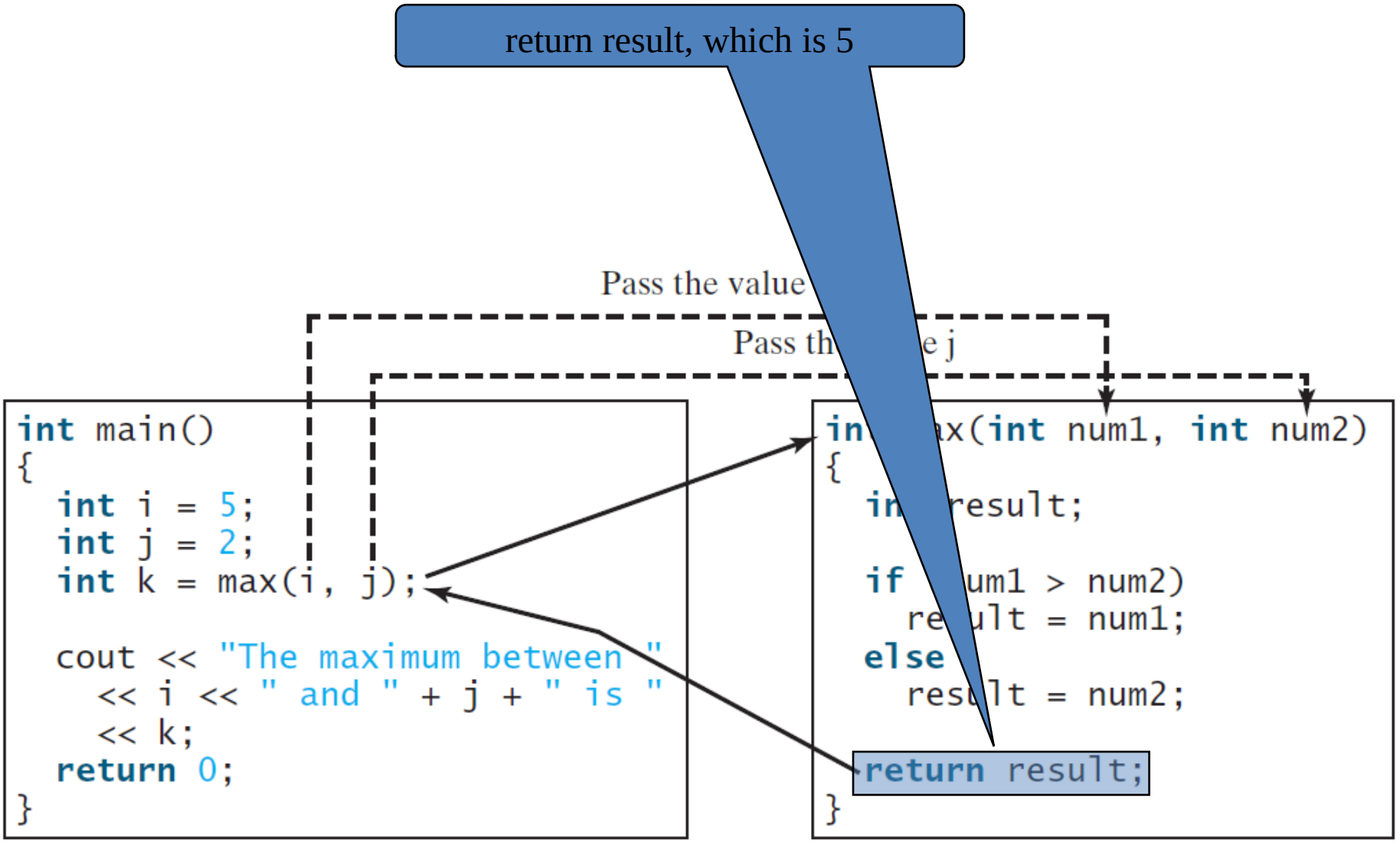
animation

Trace Function Invocation



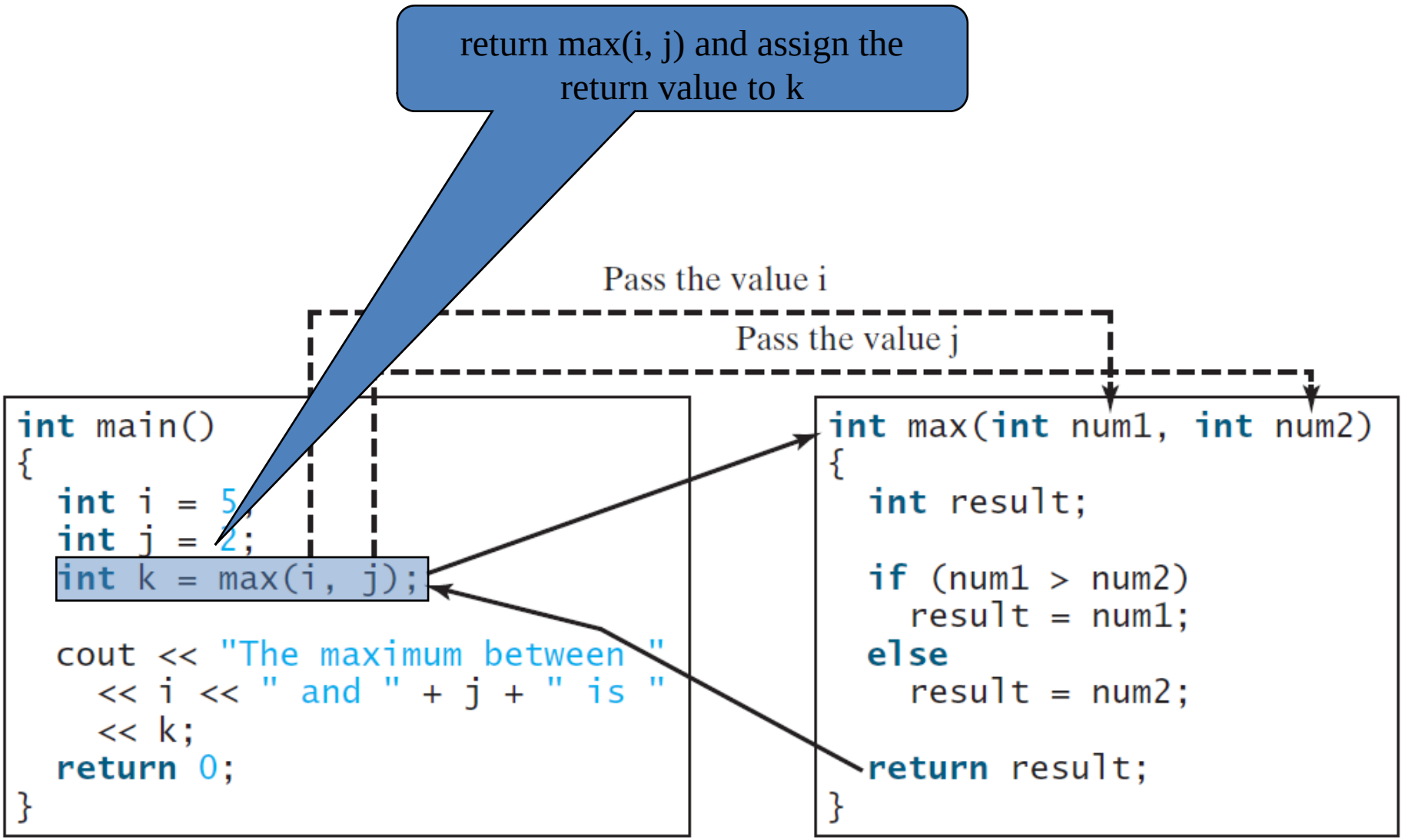
animation

Trace Function Invocation



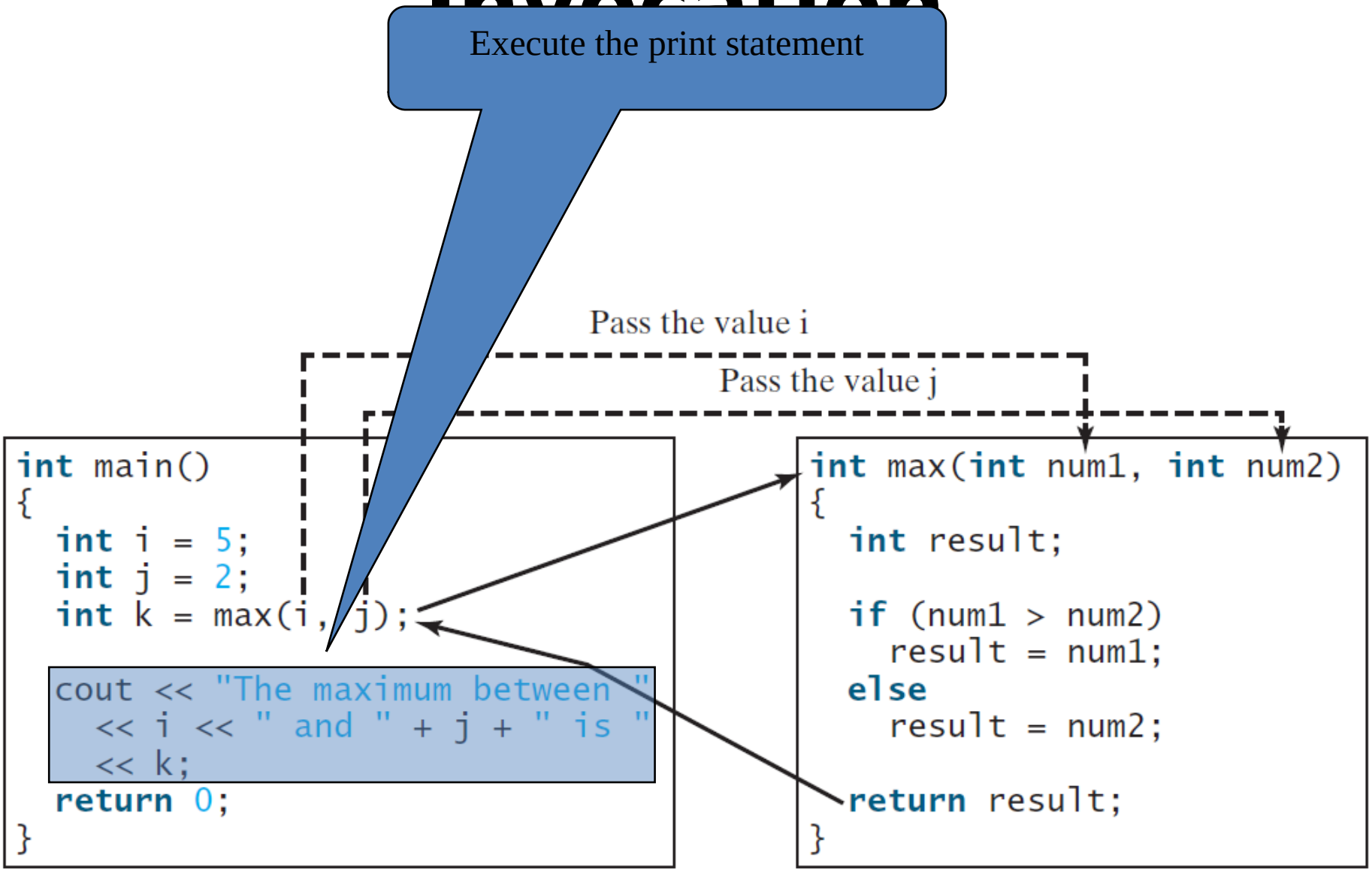
animation

Trace Function Invocation



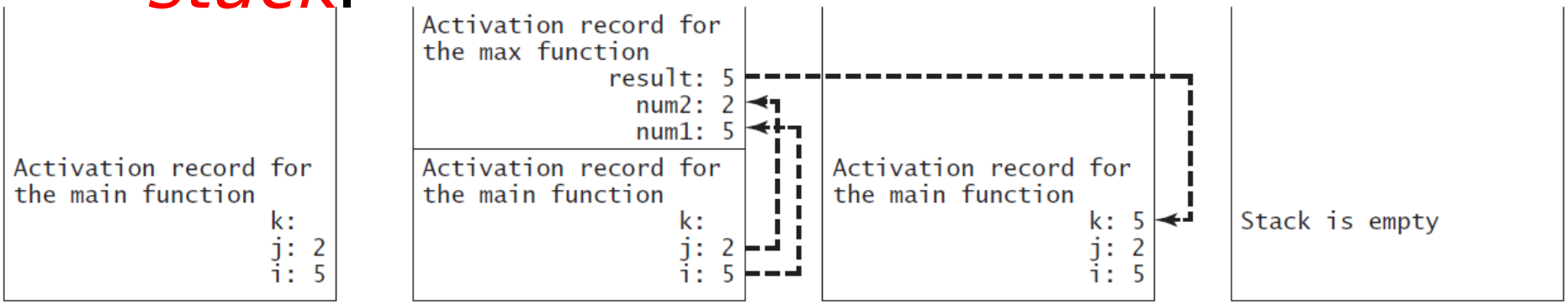
animation

Trace Function Invocation



Call Stacks

- Each time a function is invoked, the system creates an *activation record*.
- The activation record is placed in an area of memory known as a *call stack*.



(a) The main function is invoked.

(b) The max function is invoked.

(c) The max function is finished and the return value is sent to k.

(d) The main function is finished.

Outline

- Introduction
- Defining a Function
- Calling a Function
- **`void`** Functions
- Passing Arguments by Value
- Modularizing Code
- Overloading Functions
- Function Prototypes
- Default Arguments
- Inline Functions
- Local, Global, and Static Local Variables
- Passing Arguments by Reference
- Constant Reference Parameters

void Functions

A void function does not return a value.

Want to print the grade for a given score.
Two solutions:

1. **`printGrade`** prints the grade
2. **`getGrade`** prints the grade

TestVoidFunction Run

TestReturnGradeFunction Run

void Functions

```
void printGrade(double score)
{
    if (score >= 90.0)
        cout << 'A' << endl;
    else if (score >= 80.0)
        cout << 'B' << endl;
    else if (score >= 70.0)
        cout << 'C' << endl;
    else if (score >= 60.0)
        cout << 'D' << endl;
    else
        cout << 'F' << endl;
}

int main()
{
    cout << "Enter a score: ";
    double score;
    cin >> score;

    cout << "The grade is ";
    printGrade(score);
    return 0;
}
```

```
char getGrade(double score)
{
    if (score >= 90.0)
        return 'A';
    else if (score >= 80.0)
        return 'B';
    else if (score >= 70.0)
        return 'C';
    else if (score >= 60.0)
        return 'D';
    else
        return 'F';
}

int main()
{
 cout << "Enter a score: ";
    double score;
    cin >> score;

    cout << "The grade is ";
    cout << getGrade(score) <<
endl;
    return 0;
}
```

Terminating a Program

- You can terminate a program at abnormal conditions by calling **exit(n)**.
- Select the integer **n** to specify the error type.

```
void printGrade(double score)
{
    if (score < 0 || score > 100)
    {
        cout << "Invalid score" << endl;
        exit(1);
    }
    if (score >= 90.0)
        cout << 'A';
    else if (score >= 80.0)
        cout << 'B';
    else if (score >= 70.0)
        cout << 'C';
    else if (score >= 60.0)
        cout << 'D';
    else
        cout << 'F';
}
```

Outline

- Introduction
- Defining a Function
- Calling a Function
- **`void`** Functions
- Passing Arguments by Value
- Modularizing Code
- Overloading Functions
- Function Prototypes
- Default Arguments
- Inline Functions
- Local, Global, and Static Local Variables
- Passing Arguments by Reference
- Constant Reference Parameters

Passing Arguments by Value

- By default, the arguments are passed by value to parameters when invoking a function.
- When calling a function, you need to provide arguments, which must be given in the same order as their respective parameters in the function signature.
- The shown code prints **a** character **3** times.

```
void nPrint(char ch, int n)
{
    for (int i = 0; i < n; i+
+)
        cout << ch;
}

nPrint('a', 3);
```

```
aaa
```

Outline

- Introduction
- Defining a Function
- Calling a Function
- **`void`** Functions
- Passing Arguments by Value
- Modularizing Code
- Overloading Functions
- Function Prototypes
- Default Arguments
- Inline Functions
- Local, Global, and Static Local Variables
- Passing Arguments by Reference
- Constant Reference Parameters

Modularizing Code

- Modularizing makes the code easy to maintain and debug and enables the code to be reused.
- These two examples use functions to reduce complexity.

GreatestCommonDivisorFunction Run

PrimeNumberFunction Run

GreatestCommonDivisorFunction.cpp

```
int gcd(int n1, int n2)
{
    int gcd = 1; // Initial gcd is 1
    int k = 2;   // Possible gcd

    while (k <= n1 && k <= n2)
    {
        if (n1 % k == 0 && n2 % k == 0)
            gcd = k; // Update gcd
        k++;
    }
    return gcd; // Return gcd
}
int main()
{
    ...
    cout << "The greatest common divisor for " << n1 <<
        " and " << n2 << " is " << gcd(n1, n2) << endl;

    return 0;
}
```

PrimeNumberFunction.cpp 1/3

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

// Check whether number is prime
bool isPrime(int number)
{
    for (int divisor = 2; divisor <= number / 2; divisor+
+)
    {
        if (number % divisor == 0)
        {
            // If true, number is not prime
            return false; // number is not a prime
        }
    }

    return true; // number is prime
}
```

PrimeNumberFunction.cpp 2/3

```
void printPrimeNumbers(int numberOfPrimes)
{
    int count = 0; // Count the number of prime numbers
    int number = 2; // A number to be tested for primeness

    // Repeatedly find prime numbers
    while (count < numberOfPrimes)
    {
        // Print the prime number and increase the count
        if (isPrime(number))
        {
            count++; // Increase the count
            if (count % 10 == 0) // 10 numbers per line
            {
                // Print the number and advance to the new line
                cout << setw(4) << number << endl;
            }
            else
                cout << setw(4) << number;
        }
        number++; // Check if the next number is prime
    }
}
```

PrimeNumberFunction.cpp 3/3

```
int main()
{
    cout << "The first 50 prime numbers are \n";
    printPrimeNumbers(50);

    return 0;
}
```

Outline

- Introduction
- Defining a Function
- Calling a Function
- **`void`** Functions
- Passing Arguments by Value
- Modularizing Code
- Overloading Functions
- Function Prototypes
- Default Arguments
- Inline Functions
- Local, Global, and Static Local Variables
- Passing Arguments by Reference
- Constant Reference Parameters

Overloading Functions

Overloading functions enables you to define functions with the same name as long as their signatures are different.

- The **`max`** function that was used earlier works only with the **`int`** data type.
- We can define and use other max functions that accept different parameter counts and types.

TestFunctionOverloading

Run

TestFunctionOverloading.cpp 1/2

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

// Return the max between two int values
int max(int num1, int num2)
{
    if (num1 > num2)
        return num1;
    else
        return num2;
}

// Find the max between two double values
double max(double num1, double num2)
{
    if (num1 > num2)
        return num1;
    else
        return num2;
}
```

TestFunctionOverloading.cpp 2/2

```
// Return the max among three double values
double max(double num1, double num2, double num3)
{
    return max(max(num1, num2), num3);
}

int main()
{
    // Invoke the max function with int parameters
    cout << "The max between 3 and 4 is " << max(3, 4) <<
endl;

    // Invoke the max function with the double parameters
    cout << "The maximum between 3.0 and 5.4 is "
        << max(3.0, 5.4) << endl;

    // Invoke the max function with three double
parameters
    cout << "The maximum between 3.0, 5.4, and 10.14 is "
        << max(3.0, 5.4, 10.14) << endl;

    return 0;
```

Ambiguous Invocation

Sometimes there may be two or more possible matches for an invocation of a function, but the compiler cannot determine the most specific match. This is referred to as *ambiguous invocation*. Ambiguous invocation is a compilation error.

Ambiguous Invocation

```
#include <iostream>
using namespace std;
int maxNumber(int num1, double num2)
{
    if (num1 > num2)
        return num1;
    else
        return num2;
}
double maxNumber(double num1, int num2)
{
    if (num1 > num2)
        return num1;
    else
        return num2;
}
int main()
{
    cout << maxNumber(1, 2) << endl;  // Compilation
error
    return 0;
}
```

maxNumber(1.0, 2)
And
maxNumber(1, 2.0)
Are OK

Outline

- Introduction
- Defining a Function
- Calling a Function
- **`void`** Functions
- Passing Arguments by Value
- Modularizing Code
- Overloading Functions
- Function Prototypes
- Default Arguments
- Inline Functions
- Local, Global, and Static Local Variables
- Passing Arguments by Reference
- Constant Reference Parameters

Function Prototypes

- Before a function is called, it must be declared first.
- One way to ensure it is to place the declaration before all function calls.
- Another way to approach it is to declare a function prototype before the function is called.
- A *function prototype* is a function declaration without implementation.
- The implementation can be given later in the program.

TestFunctionPrototype Run

TestFunctionPrototype.cpp

```
#include <iostream>
using namespace std;
// Function prototype
int max(int num1, int num2);
double max(double num1, double num2);
double max(double num1, double num2, double num3);

int main()
{
    // Invoke the max function with int parameters
    cout << "The maximum between 3 and 4 is " <<
        max(3, 4) << endl;
    ...
}
// Return the max between two int values
int max(int num1, int num2)
{
    if (num1 > num2)
        return num1;
    else
        return num2;
}
...
```

Or simply:

```
int max(int, int);
double max(double, double);
double max(double, double,
double);
```

Outline

- Introduction
- Defining a Function
- Calling a Function
- **`void`** Functions
- Passing Arguments by Value
- Modularizing Code
- Overloading Functions
- Function Prototypes
- Default Arguments
- Inline Functions
- Local, Global, and Static Local Variables
- Passing Arguments by Reference
- Constant Reference Parameters

Default Arguments

You can define default values for parameters in a function.

The default values are passed to the parameters when a function is invoked without the arguments.

DefaultArgumentDemo Run

DefaultArgumentDemo.cpp

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

// Display area of a circle
void printArea(double radius = 1)
{
    double area = radius * radius * 3.14159;
    cout << "area is " << area << endl;
}

int main()
{
    printArea();
    printArea(4);

    return 0;
}
```

Default Arguments

- When a function contains a mixture of parameters with and without default values, those with default values must be c

```
void t1(int x, int y = 0, int z); // Illegal
void t3(int x, int y = 0, int z = 0); // Legal
```

- When an argument is left out of a function, all arguments that come after it must be

```
t3(1,  , 20); // Illegal
t3(1); // Parameters y and z are assigned a default value
```

Outline

- Introduction
- Defining a Function
- Calling a Function
- **`void`** Functions
- Passing Arguments by Value
- Modularizing Code
- Overloading Functions
- Function Prototypes
- Default Arguments
- Inline Functions
- Local, Global, and Static Local Variables
- Passing Arguments by Reference
- Constant Reference Parameters

Inline Functions

C++ provides inline functions for improving performance for short functions.

- Inline functions are not called; rather, the compiler copies the function code in line at the point of each invocation.
- To specify an inline function, precede the function declaration with the **`inline`** keyword.
- Inline functions are desirable for short functions but not for long ones.

InlineDemo

Run

InlineExpandedDemo

InlineDemo.cpp

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

inline void f(int month, int year)
{
    cout << "month is " << month << end
    cout << "year is " << year << endl;
}

int main()
{
    int month = 10, year = 2008;
    f(month, year);  // Invoke inline function
    f(9, 2010); // Invoke inline function

    return 0;
}
```

Equivalent to:

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

int main()
{
  int month = 10, year = 2008;
  cout << "month is " << month << endl;
  cout << "year is " << year << endl;
  cout << "month is " << 9 << endl;
  cout << "year is " << 2010 << endl;

  return 0;
}
```

Outline

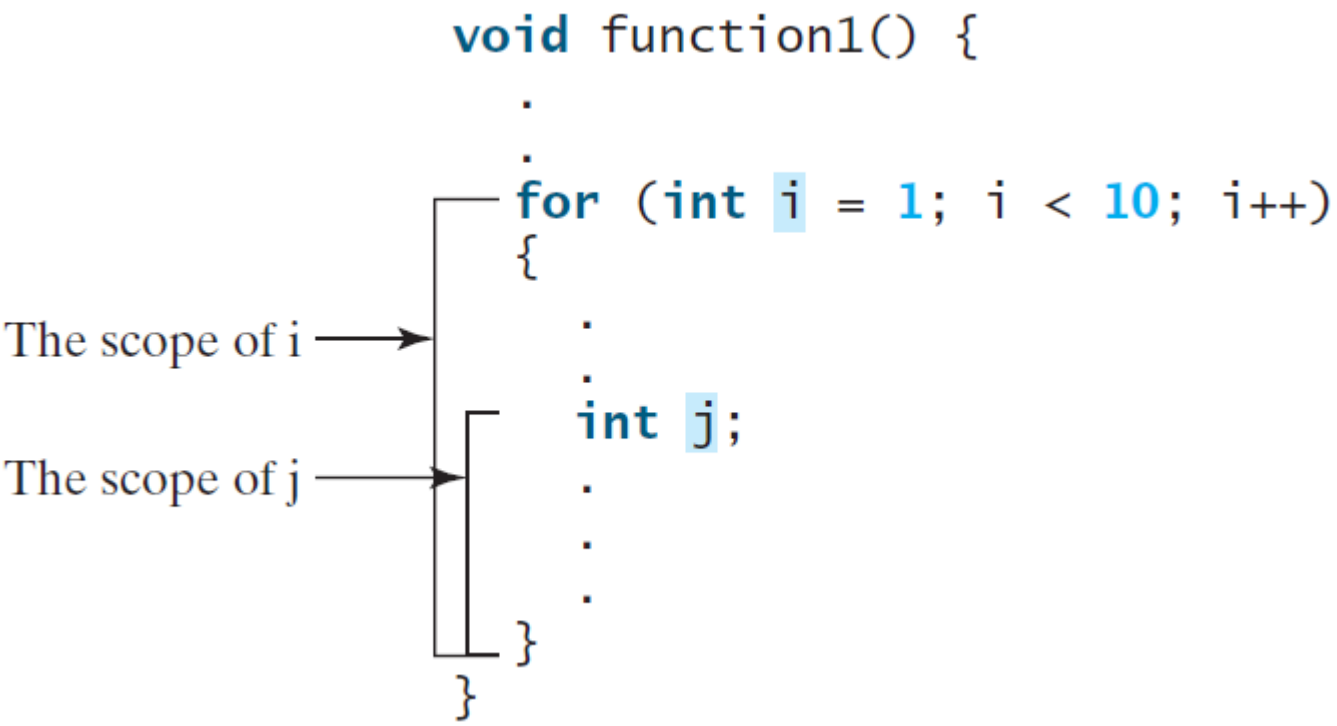
- Introduction
- Defining a Function
- Calling a Function
- **`void`** Functions
- Passing Arguments by Value
- Modularizing Code
- Overloading Functions
- Function Prototypes
- Default Arguments
- Inline Functions
- Local, Global, and Static Local Variables
- Passing Arguments by Reference
- Constant Reference Parameters

Scope of Variables

- *Scope*: the part of the program where the variable can be referenced.
- The scope of a variable starts from its declaration and continues to the end of the block that contains the variable.
- A variable can be declared as a *local*, a *global*, or a *static* local.
- A *local variable*: a variable defined inside a function.
- You can declare a local variable with the same name in different blocks.

Scope of Local Variables

- A variable declared in the initial action part of a **`for`** loop has its scope in the entire loop.
- A variable declared inside a **`for`** loop body has its scope limited the rest of the loop body.



Scope of Local Variables, cont.

- It is acceptable to declare a local variable with the same name in different non-nesting blocks.
- Avoid using same variable name in nesting blocks to minimize making mistakes.

It is fine to declare i in two nonnesting blocks

```
void function1()
{
  int x = 1;
  int y = 1;

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

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

It is not a good practice to declare i in two nesting blocks

```
void function2()
{
  int i = 1;
  int sum = 0;

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

  cout << i << endl;
  cout << sum << endl;
}
```

Global Variables

- *Global variables* are declared outside all functions and are accessible to all functions in their scope.
- Local variables do not have default values, but global variables are defaulted to zero.

VariableScopeDemo Run

VariableScopeDemo.cpp

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

void t1(); // Function
prototype
void t2(); // Function
prototype

int main()
{
    t1();
    t2();

    return 0;
}

int y; // Global variable
       // default to 0
```

```cpp
void t1()
{
    int x = 1;
    cout << "x is " << x << endl;
    cout << "y is " << y << endl;
    x++;
    y++;
}

void t2()
{
    int x = 1;
    cout << "x is " << x << endl;
    cout << "y is " << y << endl;
}
```

```
x is 1
y is 0
x is 1
y is 1
```

Unary Scope Resolution

If a local variable name is the same as a global variable name, you can access the global variable using **`::globalVariable`**. The `::` operator is known as the *unary scope resolution*.

```
#include <iostream>
using namespace std;
int v1 = 10;
int main()
{
    int v1 = 5;
    cout << "local variable v1 is " << v1 << endl;
    cout << "global variable v1 is " << ::v1 << endl;
    return 0;
}
```

```
local variable v1 is 5
global variable v1 is
10
```

Static Local Variables

- After a function completes its execution, all its local variables are destroyed.
- To retain the value stored in local variables so that they can be used in the next call, use *static local variables*.
- Static local variables are permanently allocated in the memory for the lifetime of the program.
- To declare a static variable, use the keyword **`static`**.

StaticVariableDemo Run

StaticVariableDemo.cpp

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

void t1(); // Function prototype

int main()
{
    t1();
    t1();
    return 0;
}

void t1()
{
    static int x = 1; // Static local
    int y = 1;        // Local, not static
    x++;
    y++;
    cout << "x is " << x << endl;
    cout << "y is " << y << endl;
}
```

```
x is 2
y is 2
x is 3
y is 2
```

Outline

- Introduction
- Defining a Function
- Calling a Function
- **`void`** Functions
- Passing Arguments by Value
- Modularizing Code
- Overloading Functions
- Function Prototypes
- Default Arguments
- Inline Functions
- Local, Global, and Static Local Variables
- Passing Arguments by Reference
- Constant Reference Parameters

Pass by Value

- When you invoke a function with a parameter, the value of the argument is passed to the parameter. This is referred to as *pass-by-value*.
- The variable is not affected, regardless of the changes made to the parameter inside the function.

Increment Run

Increment.cpp

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

void increment(int n)
{
    n++;
    cout << "\tn inside the function is " << n << endl;
}

int main()
{
    int x = 1;
    cout << "Before the call, x is " << x << endl;
    increment(x);
    cout << "after the call, x is " << x << endl;

    return 0;
}
```

```
Before the call, x is 1
  n inside the function is 2
after the call, x is 1
```

Reference Variables

- A *reference variable* can be used as a function parameter to reference the original variable.
- A reference variable is an alias for another variable.
- Any changes made through the reference variable are actually performed on the original variable.
- To declare a reference variable, place the ampersand (**&**) in front of the name

TestReferenceVariable Run

TestReferenceVariable.cpp

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

int main()
{
    int count = 1;
    int& r = count;
    cout << "count is " << count << endl;
    cout << "r is " << r << endl;

    r++;
    cout << "count is " << count << endl;
    cout << "r is " << r << endl;

    count = 10;
    cout << "count is " << count << endl;
    cout << "r is " << r << endl;

    return 0;
}
```

```
count is 1
r is 1
count is 2
r is 2
count is 10
r is 10
```

Pass By Reference

Parameters can be passed by reference, which makes the formal parameter an alias of the actual argument. Thus, changes made to the parameters inside the function also made to the arguments.

SwapByReference Run

```
Before invoking the swap function, num1 is 1 and num2 is 2
  Inside the swap function
  Before swapping n1 is 1 n2 is 2
  After swapping n1 is 2 n2 is 1
After invoking the swap function, num1 is 2 and num2 is 1
```

SwapByReference.cpp 1/2

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

// Swap two variables
void swap(int& n1, int& n2)
{
    cout << "\tInside the swap function" << endl;
    cout << "\tBefore swapping n1 is " << n1 <<
        " n2 is " << n2 << endl;

    // Swap n1 with n2
    int temp = n1;
    n1 = n2;
    n2 = temp;

    cout << "\tAfter swapping n1 is " << n1 <<
        " n2 is " << n2 << endl;
}
```

SwapByReference.cpp 2/2

```
int main()
{
    // Declare and initialize variables
    int num1 = 1;
    int num2 = 2;

    cout << "Before invoking the swap function, num1 is "
        << num1 << " and num2 is " << num2 << endl;

    // Invoke the swap function to attempt to swap two
variables
    swap(num1, num2);

    cout << "After invoking the swap function, num1 is "
<< num1
         << " and num2 is " << num2 << endl;

    return 0;
}
```

Pass-by-Value vs. Pass-by-Reference

- In *pass-by-value*, the actual parameter and its formal parameter are independent variables.
- In *pass-by-reference*, the actual parameter and its formal parameter refer to the same variable.
- *Pass-by-reference* is more efficient than *pass-by-value*. However, the difference is negligible for parameters of primitive types such as **`int`** and **`double`**.
- So, if a primitive data type parameter is not changed in the function, you should declare it as *pass-by-value* parameter.

Outline

- Introduction
- Defining a Function
- Calling a Function
- **`void`** Functions
- Passing Arguments by Value
- Modularizing Code
- Overloading Functions
- Function Prototypes
- Default Arguments
- Inline Functions
- Local, Global, and Static Local Variables
- Passing Arguments by Reference
- Constant Reference Parameters

Constant Reference Parameters

You can specify a constant reference parameter to prevent its value from being changed by accident.

```
// Return the max between two numbers
int max(const int& num1, const int& num2)
{
    int result;
    if (num1 > num2)
        result = num1;
    else
        result = num2;
    return result;
}
```

Outline

- Introduction
- Defining a Function
- Calling a Function
- **`void`** Functions
- Passing Arguments by Value
- Modularizing Code
- Overloading Functions
- Function Prototypes
- Default Arguments
- Inline Functions
- Local, Global, and Static Local Variables
- Passing Arguments by Reference
- Constant Reference Parameters