Chapter 4-6

Jeff Davis

ECE2036

Topics You Are Responsible For

- 4.5 if Selection Statement
- 4.6 if..else Double Selection Statement
  - Conditional Operator (? :)
  - else if
- 4.7 while Repetition Statement
- 4.11 Assignment Operators
- 4.12 Increment and Decrement Operators

Topics You Are Responsible For

- 5.3 for Repetition Statement
- 5.4 Examples Using the for Statement
- 5.8 Logical Operators
- 5.9 Confusing the Equality (==) and Assignment (=) Operators

Topics You Are Responsible For

- Chapter 5.5 : do..while Repetition Statements
  - rand(); and srand(seed); highlighted
- Chapter 5.3 : for Repetition Statement
  - part I: local variable hides global with same name
- Chapter 5.7 : break and continue statements
- Chapter 6.1 : Introduction
- Chapter 6.2 : Program Components in C++
- Chapter 6.3 : Math Library Functions
- Chapter 6.4 : Function Definitions with Multiple Parameters
- Chapter 6.9 : Storage Classes
- Chapter 6.10 : Scope Rules

Topics You Are Responsible For

- Chapter 6.4 : Function Definitions with Multiple Parameters
- Chapter 6.9 : Storage Classes
- Chapter 6.10 : Scope Rules
- Chapter 6.11 : Function Call Stack and Activation Records
- Chapter 6.12 : Functions with Empty Parameter Lists
- Chapter 6.13 : Inline Functions
- Chapter 6.14 : References and Reference Parameters
- Chapter 6.15 : Default Arguments
- Chapter 6.16 : Unary Scope Resolution Operator
- Chapter 6.17: Function Overloading
- Chapter 6.18: Function Templates
- Chapter 6.19-6.21: Recursion and Examples

Order of Precedence for Operators

Precedence	Operator	Description	Associativity
1	`::`	Scope resolution	Left-to-right

2	`++` `--`
	`()`
	`[]`
	`.`
	`->`
		Suffix/postfix increment and decrement
		Function call
		Array subscripting
		Element selection by reference
		Element selection through pointer	

3	`++` `--`
	`+` `-`
	`!` `~`
	`(`*type*`)`
	`*`
	`&`
	`sizeof`
	`new`, `new[]`
	`delete`, `delete[]`
		Prefix increment and decrement
		Unary plus and minus
		Logical NOT and bitwise NOT
		Type cast
		Indirection (dereference)
		Address-of
		Size-of
		Dynamic memory allocation
		Dynamic memory deallocation
			Right-to-left

4	`.*` `->*`	Pointer to member	Left-to-right
5	`*` `/` `%`	Multiplication, division, and remainder	
6	`+` `-`	Addition and subtraction	
7	`<<` `>>`	Bitwise left shift and right shift	
8	`<` `<=` `>` `>=`	For relational operators < and ≤ respectively For relational operators > and ≥ respectively	
9	`==` `!=`	For relational = and ≠ respectively	
10	`&`	Bitwise AND	
11	`^`	Bitwise XOR (exclusive or)	
12	`|`	Bitwise OR (inclusive or)	
13	`&&`	Logical AND	
14	`||`	Logical OR	

15	`?:`
	`=`
	`+=` `-=`
	`*=` `/=` `%=`
	`<<=` `>>=`
	`&=` `^=` `|=`
		Ternary conditional[1]
		Direct assignment (provided by default for C++ classes)
		Assignment by sum and difference
		Assignment by product, quotient, and remainder
		Assignment by bitwise left shift and right shift
		Assignment by bitwise AND, XOR, and OR
			Right-to-left

16	`throw`	Throw operator (for exceptions)	
17	`,`	Comma	Left-to-right

Switch Statements

```
switch ( variable)
{
	case value1:

		//statements1;
		break;

	case value2:

		//statements2;
		break;

	default:
		//default statements


}
```

switch to Bash shell to implement the program switchStatementExample.cpp

5.5 do..while loop structure

do..while loop

```
do
{

//body to be implemented

} while (condition);
```

implements body at least once!

checks last

while loop

```
while (condition)
{

//body to be implemented

}
```

checks first

implements body if condition is true!

5.5 do..while loop structure

<u>Example of Guessing a Random Number!</u>

- User is prompted for a guess between 1 to 6
- Error checking is performed
- rand() and srand(); is highlighted
- include <cstdlib>

switch to Bash shell to implement the program
doWhileLoopExample.cpp

5.5 for loop structure

keyword

Control variable name

Controls number of loop iterations, which in this case is 10 times

Don’t put semi-colon here!

```
for ( counter = 1; counter <=10; counter++)
{
//body to be implemented
}
```

counter increment

5.5 for loop structure

Variable Scope Issues!

for loop

```
int main()
{

int counter;

...

for ( counter = 1; counter <=10; counter++)
{
   //body to be implemented
}

}//end main
```

for loop – counter scope only in loop!

```
for (int counter =1; counter <=10; counter ++)
{
   //body to be implemented
}
```

counter only in scope inside for loop for this condition.

switch to Bash shell to implement the program forLoopandScope.cpp

Declaring Variables Inside a Loop

```
for (int i = 0; i<N; i++)
{
   int temp;
   // temp is uninitialized

   // temp will have the same
   //value at the end of previous loop

   //perform some action here

}//end of for loop
```

```
for (int i = 0; i<N; i++)
{
   int temp =0;
   // temp will be reinitialized at the
   //beginning of the loop each time.

   //perform some action here

}//end of for loop
```

Both temp and i cannot be used outside the loop block!!!

5.7 break and continue statements

```
int counter;
int sum =0;
...
//hidden statements manipulating sum
...
for ( counter = 1; counter <=10; counter++)
{

  if (sum >= 50)
    break; // this gets out of the loop

  if (sum == 25)
    continue; // gets out of JUST this iteration of loop

  sum = sum + counter;

} //end of for loop
```

break and continue are typically not consider good programming practice

6.1 Introduction

"start simple and evolve to the complex"

↓

divide and conquer!

↓

"creating functions helps in this task"

↓

Software Engineering Observation 6.1 : "To promote software reusability, every function should be limited to performing a single, well-define task, and the name of the function should express that task effectively!"

6.2 Program Components in C++

functions utility

divide and conquer!

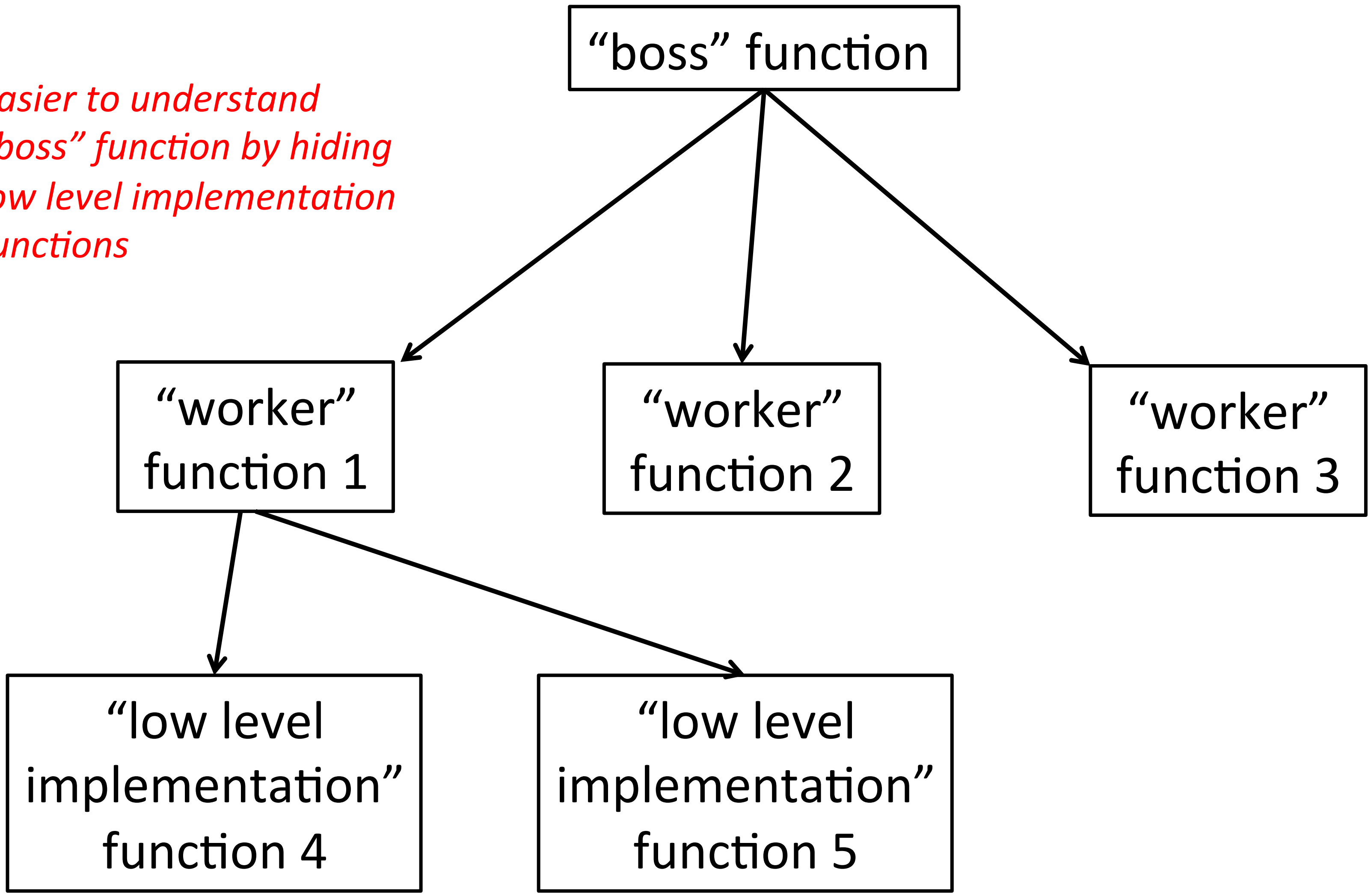
code reuse!

hiding implementation

why is this a good thing?

6.2 Program Components in C++

easier to understand "boss" function by hiding low level implementation functions



6.3 Math Library Functions

standard libraries in c

To use in c++ you typically add 'c' to the library name!!

c

```
#include <math.h>
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
```

c++

```
#include <cmath>
#include <ctime>
#include <cstdlib>
#include <cstdio>
```

6.3 Math Library Functions

#include <cmath>

ceil(x)

shift right on number line!!

ceil (9.2) gives 10.0

ceil (-9.8) gives -9.0

floor(x)

shift left on number line!!

floor (9.8) gives 9.0

floor (-9.8) gives -10.0

fabs(x)

fabs(9.8) gives 9.8

fabs(-9.8) gives 9.8

```
double round ( double number)
{
     return (  number >=0 ? int (number +0.5) : int (number-0.5) );
```

switch to Bash shell to implement the program roundExample.cpp

6.3 Math Library Functions

#include <cmath>

cos(x)
sin(x) ← x is in radians – NOT degrees
tan(x)

exp(x)
log(x) ← this is natural logarithm (base e)
log10(x) ← this is logarithm (base 10)

pow(x, y) ← x raised to the y (x^y)
sqrt(x)

6.4 Function Definitions

If the arguments passed to a function do not match the types specified in the function's prototype, the compiler attempts to convert the arguments to those types.

This is call argument coercion!!

6.4 Function Definitions

Example of Argument Coercion!!

Example: C++'s will try to convert values

```
int add_three_int(int, int, int); //function prototype

  ...

double num1,num2,num3;  //variable declaration in main

  ...

cout << add_three_int(num1, num2, num3) << endl;
```

compiler will convert to int values without warning!

switch to Bash shell to implement the program argument_coercion.cpp

6.4 Function Definitions

Promotion Hierarchy for Fundamental Data Types

highest

- long double
- double
- float
- unsigned long long int
- long long int
- unsigned long int
- long int
- unsigned int
- int
- unsigned short int
- short int
- unsigned char
- char
- bool

lowest

int promoted to double … typically okay

double promoted to int… could be be issues!

switch to Bash shell to implement the program sizeoftypes.cpp

6.4 Function Definitions

What does compiler do with “mixed-expressions”

```
doubleResult = intNumber + doubleNumber + charVariable;
```

promoted to double

The type of each value in a mixed-type expression is promoted to the “highest” type in the expression.

switch to Bash shell to implement the program: mixedExpression.cpp

This upgrade occurs for a given operation between mixed types – show ExampleDivision.cc

6.9 Storage Class

Common Variable Attributes

name

int number = 80;

type

value

size: 32 bits

6.9 Storage Class

<u>More Variable Attributes</u>

Storage Class

"Hold long does it exists in memory?"

Scope

"Where can it be accessed?"

Linkage

"Can other linked files use it?"

6.9 Storage Class

<u>Storage Class Categories</u>

Automatic Storage Class

```
auto double exampleDouble;

register int exampleInt;
```

Static Storage Class

```
extern double exampleDouble;

static int exampleInt;
```

6.9 Storage Class

Automatic Storage Class

"These variables are created when a program enters a block in which they are defined AND are destroyed when the program leaves the block!

```
auto double exampleDouble;
```

This is the default storage class and the keyword is not typically used

```
register int exampleInt;
```

This forces the compiler to keep the variable in a register (if available)! → ***The key advantage is SPEED! Especially if the variable is used a lot.. Like a counter.***

6.9 Storage Class

Static Storage Class Categories

"Static-Storage-class variables exist in memory from the point at which the program begins execution and lasts for the duration of the program"

```
double globalDouble;

int main()
{
}
```

global variables are accessed by functions in the same file!

```
extern double globalDouble;
```

The 'extern' keyword is added to definition in OTHER files that a global is to be used!

```
static int exampleInt;
```

- The 'static' keyword is added to local variables in functions to insure they are not erased periodically.
- Scoping is still within the block it is defined!

6.10 Scope Rules

Let's examine this by example

-global scope

-function scope

-local scope

ScopeExample1.cpp → example of global scope

ScopeExample2.cpp → example with static scope

6.13 Inline Functions

Let's return to our example round

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

inline double round( const double num)
{
  return( num >=0 ? (int) (num+0.5) : (int) (num-0.5));
}

int main()
{

double input_number;

cout << "Please input a number to be rounded: " ;
cin >> input_number;

cout << "The number " << input_number << " rounded is = " << round(input_number) <<endl;
```

must have inline function definition before it is used in the function (i.e. not just function prototype).

Compiler makes an actual copy of the code inside the program

Why would you want to do this? → Could be a faster!

6.14 References

Two ways to pass arguments to a function

pass-by-value

copy is made of variable
AND ORIGINAL IS NOT CHANGED!

(Note that this could effect performance and memory of your program)

pass-by-reference

Essentially you are passing the address to the actual value so IT CAN BE CHANGED BY THE FUNCTION!

(Note that this could cause accidental change to a variable that you did not intend)

6.14 References

Call by value and reference with round function!

```
double roundByValue(double num)
{
  num = ( num >=0 ? (int) (num+0.5) : (int) (num-0.5));
  return (num);
}

void roundByReference(double &num)
{
  num = ( num >=0 ? (int) (num+0.5) : (int) (num-0.5));
  return;
}
```

This reference is called an "alias" of the variable!

Example of calls in main function

```
cout << roundByValue(input_number) << endl;
```

```
roundByReference(input_number);
```

example code: Pass_by_value_reference_example.cpp

6.14 Pass by reference

- Argument coercion will not apply to variable passed by reference!
  - Show example
- You can not pass constant values if passed by reference… (no memory address)
- You can not pass expressions by reference.

6.15 Default Arguments

Can arguments to functions have default values if they are not specified by the client?

Said another way!

Can functions have optional arguments?

ANSWER IS YES!!!

6.15 Default Arguments

Requirements and Syntax

1. Default arguments must be RIGHT MOST argument in parameter list!

```
int boxVolume (int length, int width, int height)
```

2. The default variable are specified in the function prototype!

```
int boxVolume (int = 1, int = 1, int = 1);
```

3. If an argument is omitted in function call THEN ALL PARAMETERS TO THE RIGHT ALSO MUST BE OMITTED!!!!

`boxVolume()` `boxVolume(3)` `boxVolume(3,4)` `boxVolume(3,4,2)`

example code: BoxVolume.cpp

Type Casting

"You can explicitly force a change in the fundamental types and the compiler will do conversion"

- long double
- double
- float
- unsigned long long int
- long long int
- unsigned long int
- long int
- unsigned int
- int
- unsigned short int
- short int
- unsigned char
- char
- bool

<u>Syntax</u>

(type you want to change to) variable_name

<u>Examples</u>

```
cout << ( int) doubleNumber << endl;
```

```
cout << (float) intNum1/intNum2 << endl;
```

example code: TypeCastingCode.cpp

6.16 Unary Scope Resolution Operator

You can have local and global variables with the same name AND access always the global variable by using the "Unary Scope Resolution Operator"

"Unary Scope Resolution Operator"

```
#include <iostream>
using namespace std;
int number = 7;  // This is a global definition
int main()
{
int number = 10;
cout  << "The local value is " << number <<endl;
cout << "The global value is " << ::number << endl;
return 0;
}
```

This is like a default namespace for globally defined variables!

example code: UnaryScopeResolutionOperator.cpp

6.16 Unary Scope Resolution Operator

Good Programming Practice 6.4

"Always using the unary scope resolution operator (::) to refer to global variables makes programs easier to read and understand because it makes it clear that you're intending to access a global variable rather than a non-global variable."

Error-Prevention Tip 6.5

"Avoid using variables of the same name for different purposes in a program. Although this is allowed in various circumstances, it can lead to errors."

6.17 Function Overloading

"You can have functions with the same name as long as they have different parameter lists! "

C++ compiler picks the CORRECT function based on the number, types and order of the arguments in the call.

Typically function overloading is used to create several functions that perform the SIMILAR TASKS with DIFFERENT data types.

6.17 Function Overloading

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

int cube(int);
double cube(double);
float cube (float);

int main()
{
double doubleVar = 7.8;
float floatVar = 2.3;
int intVar = 2;

cout << "Double call gives "<< cube(doubleVar) << " and size is " << sizeof(doubleVar) << " bytes" << endl;
cout << "Float call gives "<< cube(floatVar) << " and size is " << sizeof(floatVar) << " bytes" << endl;
cout << "Int call gives "<< cube(intVar) << " and size is " << sizeof(intVar) << " bytes" << endl;

}//end of main

int cube (int num)
{   return( num*num*num); }

double cube (double num)
{ return( num*num*num); }

float cube (float num)
{  return( num*num*num);}
```

same function name but different parameter list!!!

example code: FunctionOverloadExample.cpp

6.17 Function Overloading

Common Programming Error 6.11

"Creating overloaded functions with identical parameter lists and different return types is a compilation error."

Common Programming Error 6.12

"A function with *default arguments* omitted might be called identically to another overloaded functions: this will cause a compilation error."

6.18 Function Templates

“Overloaded functions are normally used to perform SIMILAR OVERALL OPERATION that involve different program logic on different data types.”

↓

“If the program logic and operations are IDENTICAL for each data type, overloading may be performed more compactly and conveniently by using **function templates**.”

6.18 Function Templates

keywords

typically in a header file before the actual function call… (i.e. no function prototypes)

```
template < typename T >  // or template< typename T >
T maximum( T value1, T value2, T value3 )
{
   T maximumValue = value1; // assume value1 is maximum

   // determine whether value2 is greater than maximumValue
   if ( value2 > maximumValue )
      maximumValue = value2;

   // determine whether value3 is greater than maximumValue
   if ( value3 > maximumValue )
      maximumValue = value3;

   return maximumValue;
} // end function template maximum
```

<u>Calling the function</u>

```
cout << maximum (intnum1, intnum2, intnum3) <<endl
```

6.18 Function Templates

Calling the function

Method 1

```
cout << maximum (intnum1, intnum2, intnum3) <<endl;
cout << maximum (doublenum1, doublenum2, doublenum3) <<endl;
```

Method 2

```
cout << maximum <int> (intnum1, intnum2, intnum3) << endl;
cout << maximum <double> (doublenum1, doublenum2, doublenum3) <<endl;
```

6.18 Function Templates

Example 2: Changing Cubic Overloading Functions

```
template <typename T>
T cube (T num)
{
  return( num*num*num);
}

int main()
{
double doubleVar = 7.8;
float floatVar = 2.3;
int intVar = 2;

cout << "Double call gives "<< cube(doubleVar) << endl;
cout << "Float call gives "<< cube(floatVar) << << endl;
cout << "Int call gives "<< cube(intVar) << endl;

}//end of main
```

example code: CubicTemplate.cpp

6.18 Function Templates

Example 3: Multiple Arguments

```
//This includes the example of a template  -- maybe show modification in class
#include <iostream>
using namespace std;

template <typename T1, typename T2, typename T3>
T1 mult(T2 num1, T3 num2)
{
  return( num1 *num2);
}

int main()
{
double doubleVar = 7.8;
float floatVar = 2.3;
int intVar = 2;

cout << "Return Int with first argument double and second argument float " << endl;
cout << mult<int,double,float> (doubleVar,floatVar) << endl;
//cout << mult<int>(doubleVar,floatVar) << endl;
//cout << mult<double>(doubleVar,floatVar) << endl;

return (0);

}//end of main
```


example: multTemplate.cpp

Function Templates
vs.
Function Overloads

Let’s conclude with idea of “function call stack”

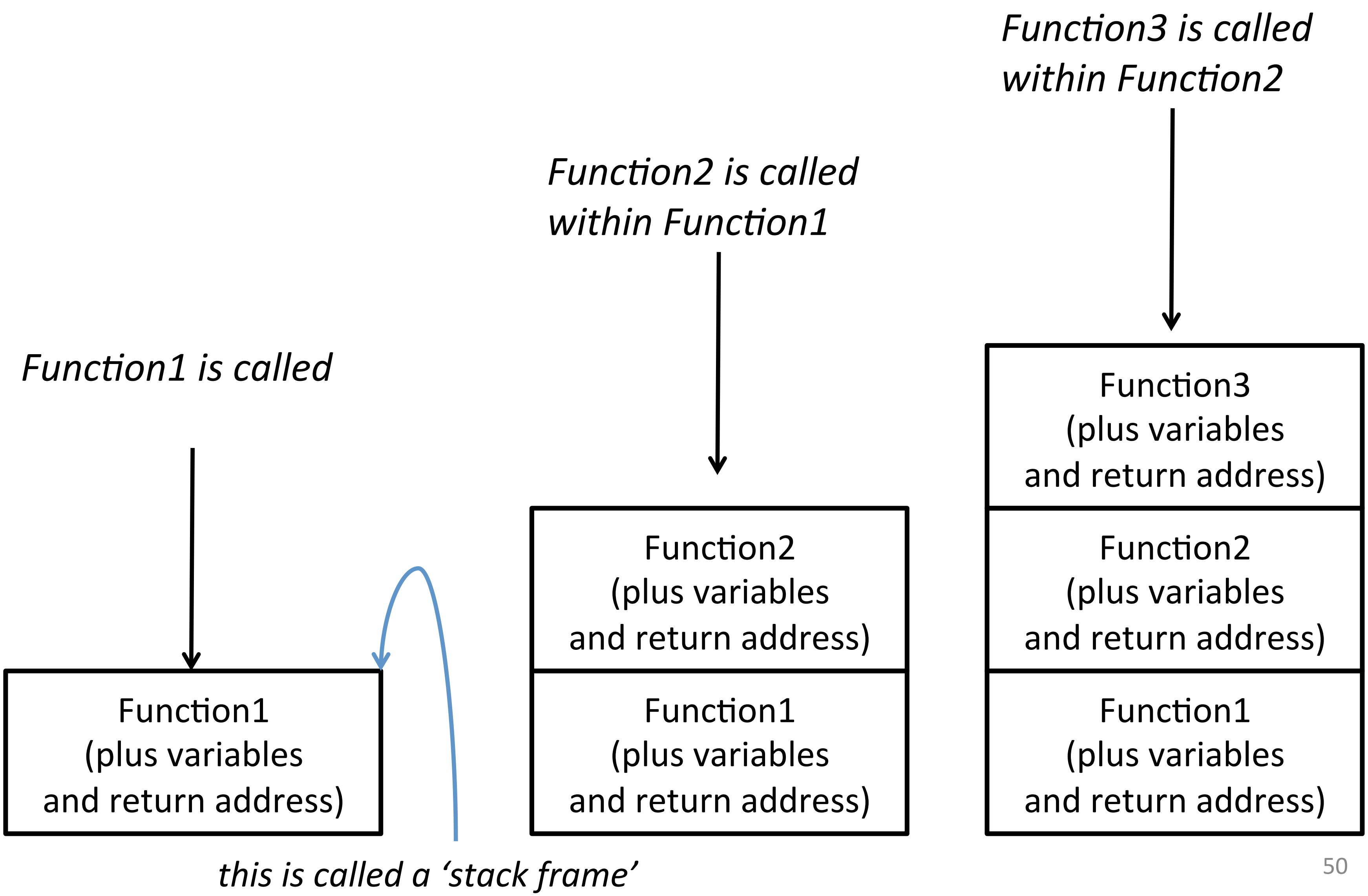
What is a Stack?

“push” plates **on** the stack

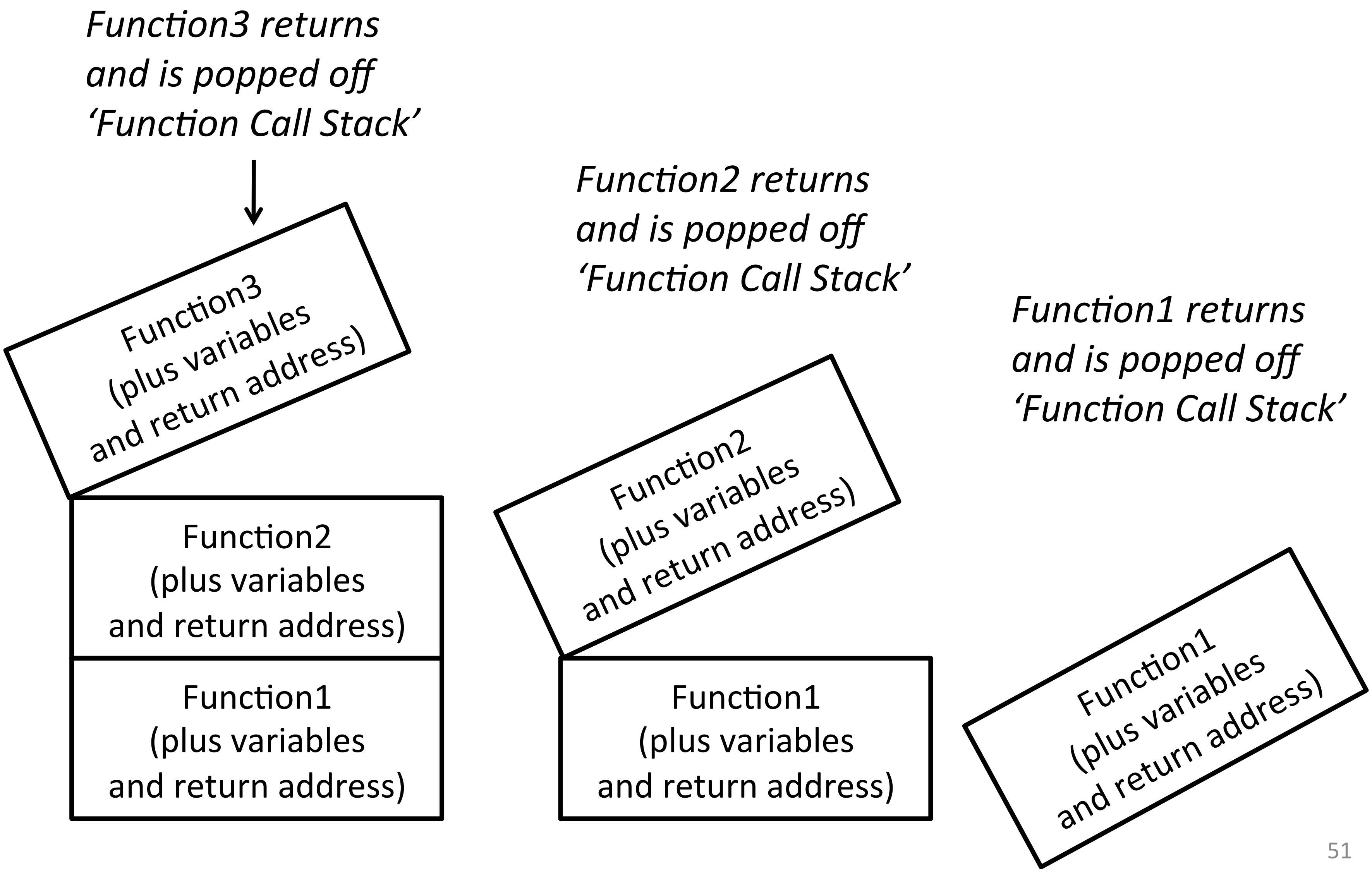
Last-In-First-Out (LIFO)

“pop” plates **off** the stack

Function Call Stack



Function Call Stack



```cpp
// Fig. 6.14: fig06_14.cpp
// square function used to demonstrate the function
// call stack and activation records.
#include <iostream>
using namespace std;

int square( int ); // prototype for function square

int main()
{
   int a = 10; // value to square (local automatic variable in main)

   cout << a << " squared: " << square( a ) << endl; // display a squared
} // end main

// returns the square of an integer
int square( int x ) // x is a local variable
{
   return x * x; // calculate square and return result
} // end function square
```

Fig. 6.14 | `square` function used to demonstrate the function call stack and activation records. (Part 1 of 2.)

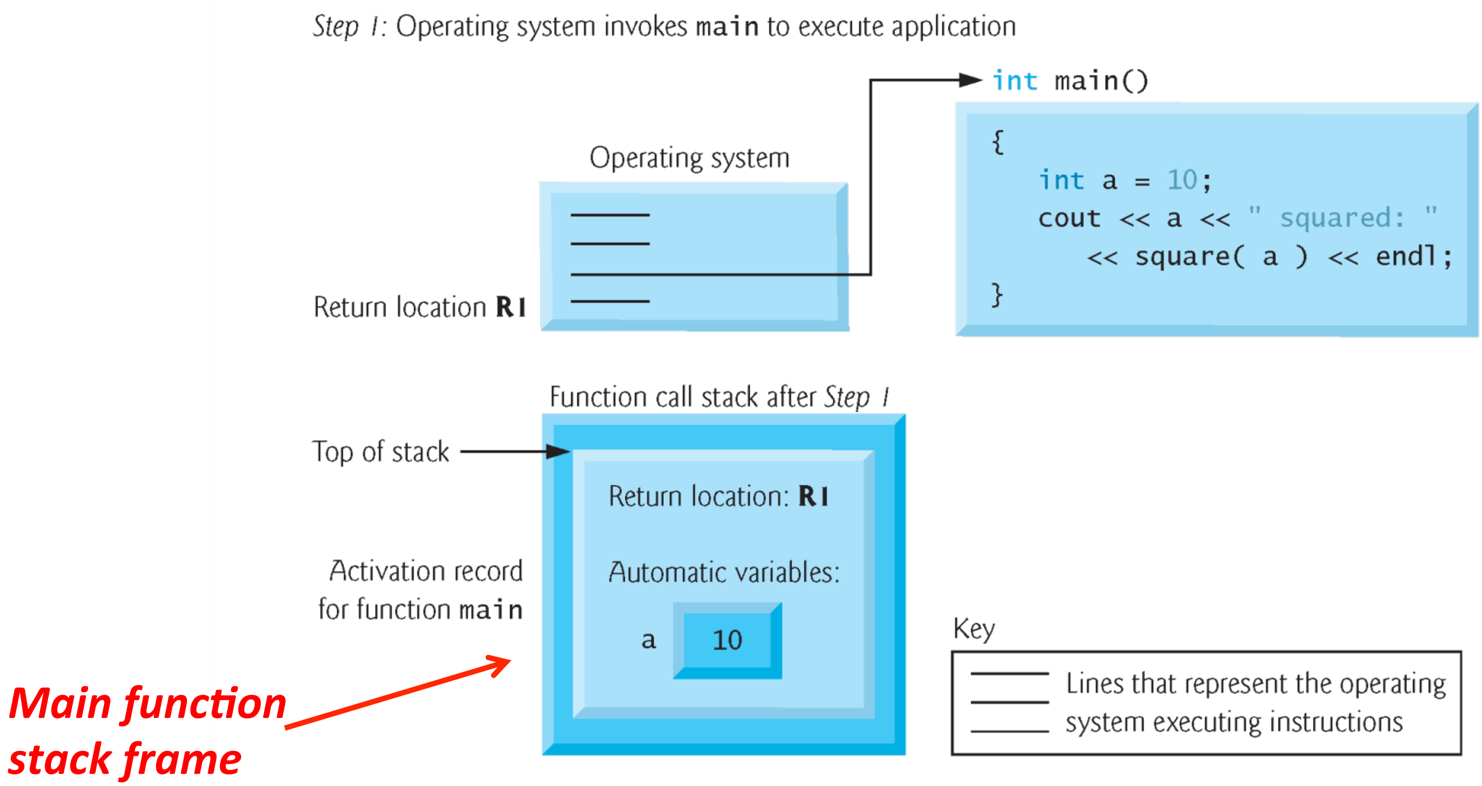
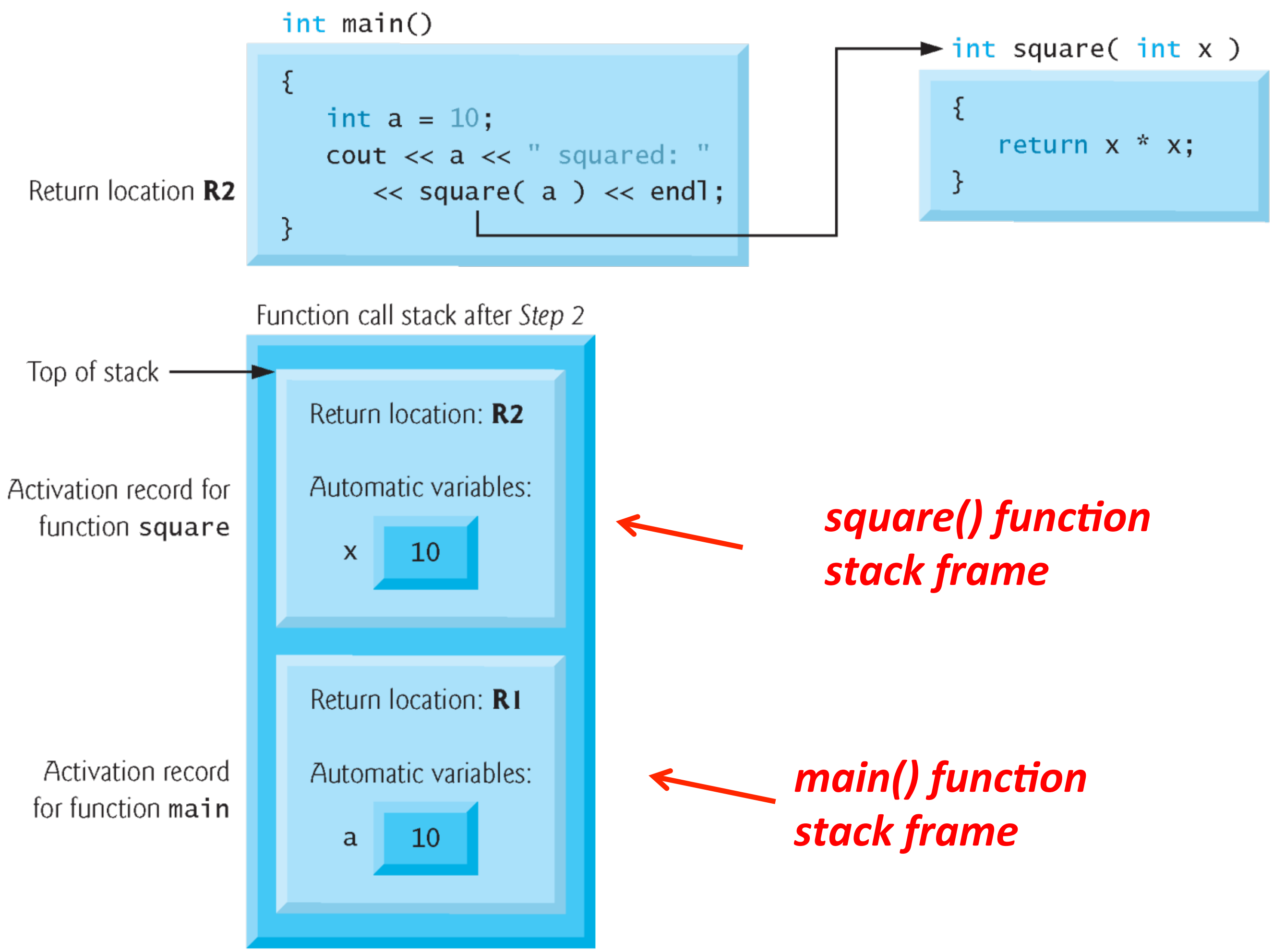


Fig. 6.15 | Function call stack after the operating system invokes `main` to execute the program.

The entry pushed onto the stack is called the "Stack Frame"

Step 2: `main` invokes function `square` to perform calculation



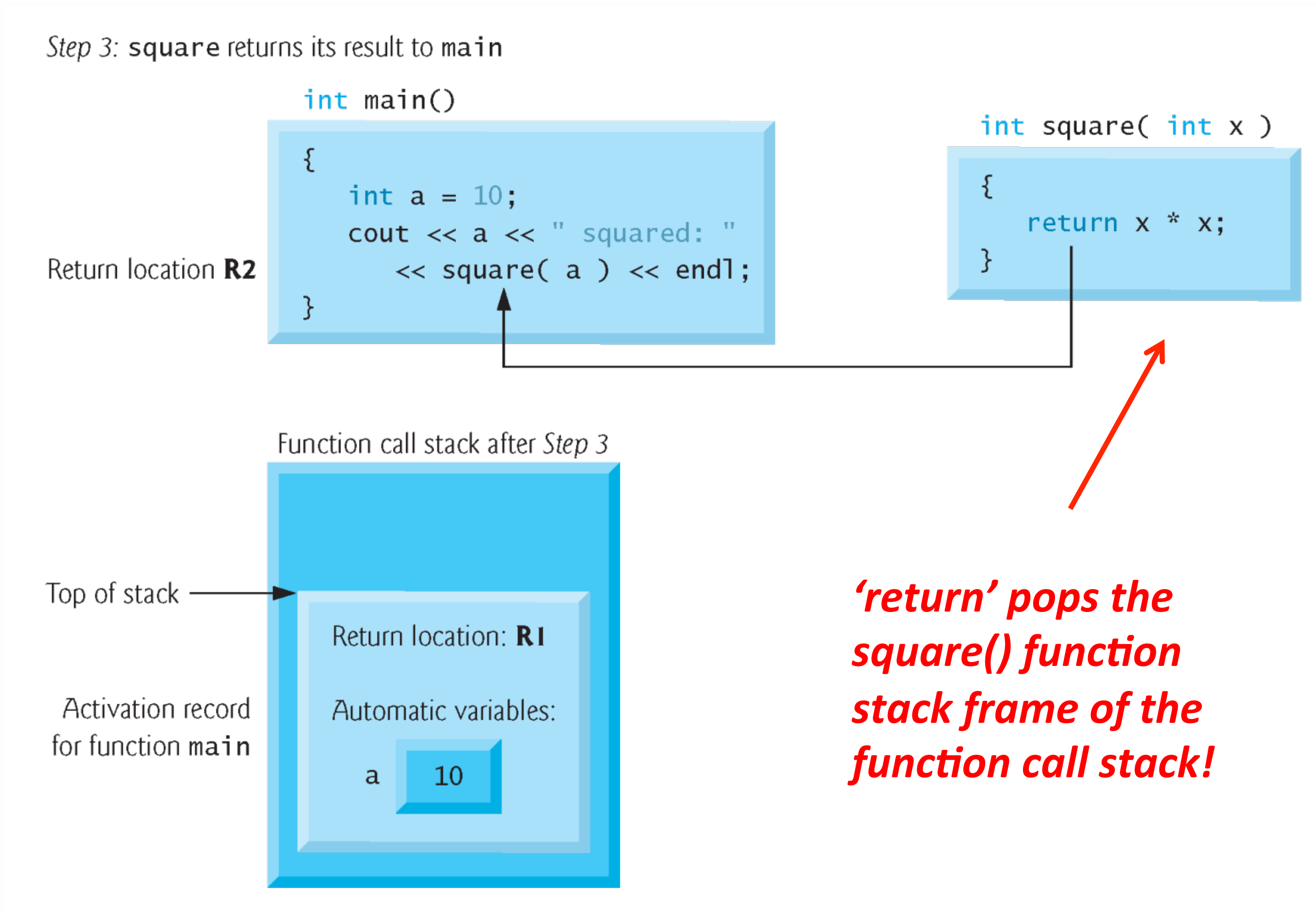


Fig. 6.17 | Function call stack after function `square` returns to `main`.

'return' pops the square() function stack frame of the function call stack!

6.19 – 6.21 Recursion

"Any function that calls ITSELF is referred to as a recursive function!!"

Factorial : n! = (n)(n-1)(n-2)…*1

```
unsigned long int factorial( int);

int main()
{
 int number;
 cout << "Please input your number : " ;
 cin >> number;
 cout << "The factorial of this number is: " << factorial(number) << endl;
 return 0;
}

unsigned long int factorial(int num)
{ // assumes num > 1
  if (num == 1)
    return 1;
  else
    return( factorial(num-1)*num);
}
```

6.19 – 6.21 Recursion

Recursive function calls push values on stack!

push each function call on a stack

factorial(1)

factorial(2)

factorial(3)

factorial(4)

factorial(5)

once the value is resolved it pops off the stack!

factorial(1) (1)

factorial(2)

factorial(3)

factorial(4)

factorial(5)

factorial(2) (2)

factorial(3)

factorial(4)

factorial(5)

factorial(3) (6)

factorial(4)

factorial(5)

6.19 – 6.21 Recursion

Recursive Algorithm

```
unsigned long int factorial(int num)
{ //assumes  num > 0

  if (num == 1)
    return 1;
  else
    return( factorial(num-1)*num);

}
```

example code: FactorialExample.cpp

Iterative Algorithm

```
unsigned long int factorial_iterative(int number)
{ //assumes number > 0
  unsigned long int total = 1;
  int I;
  for (i = number; i >= 1 ; i--)
  {
    total  *= i;
  }
  return (total);
}// end factorial_iterative
~
```

example code: IterativeSolution.cpp

6.19 – 6.21 Recursion

Software Engineering Observation 6.14

"Any problem that can be solved recursively can also be solved iteratively. A recursive approach is normally chosen when the recursive approach more naturally mirrors the problem and results in a program that's easier to understand and debug. Another reason to choose a recursive solution is that an iterative solution is not apparent."

Performance Tip 6.8

"Avoid using recursion in performance situations. Recursive calls take time and consume additional memory"

Let's look at an example where this can be a problem.

6.19 – 6.21 Recursion

- Classic Example: Fibonacci Series

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89…

The n^{th} number is $(n-1)^{th}$ number plus the (n-2)th number

6.19 – 6.21 Recursion

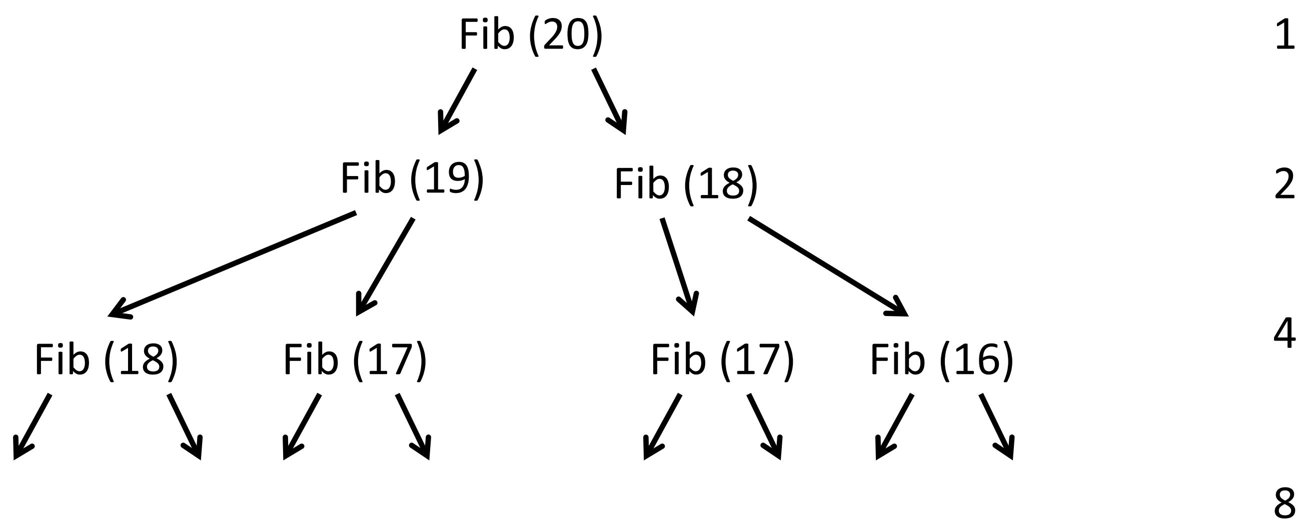
Example: Calculate the ratio of the nth number to the (n-1)th number in the Fibonacci Series.

```
long int Fibonacci(int num)
{

  switch(num)
  {
    case 1: return(0);
    case 2: return(1);
    default: return (Fibonacci(num-2) + Fibonacci(num-1));
  }
}
```

example code: FibonacciRatio.cpp

6.19 – 6.21 Recursion



This has exponential complexity!!! $\sim 2^{20}$

About one million calls to perform what we perform with 20 additions!

6.19 – 6.21 Recursion

Performance Tip 6.7
"Avoid Fibonacci-style recursive programs that result in an exponential explosion of function calls!"

Run code FibonacciRatio.cpp -- discuss golden ratio!