CS2310 Computer Programming

LT05: Function

Computer Science, City University of Hong Kong

Semester A 2023-24

About Midterm

- Week 8 lecture time (27/10 Friday, 12:00)
- In classroom, on paper
  - Formal proof needed for request of absence
- One hour
- From Lec 1 (Intro) to Lec 6 (Array)
- 15% of final mark
- Sample questions and detailed announcement will be released shortly

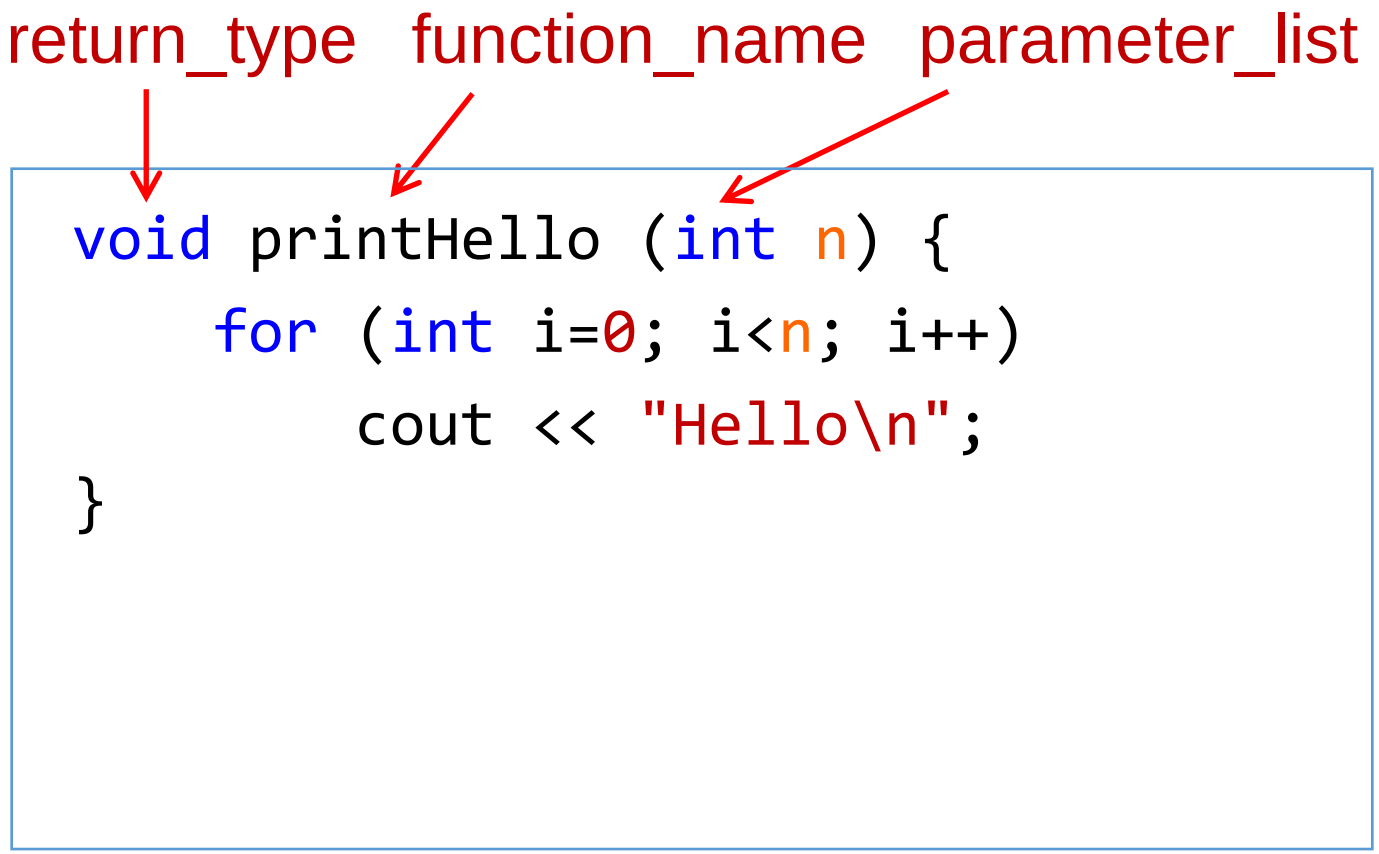
What's Function?

- A collection of statements that perform a specific task
- Functions are used to break a problem down into manageable pieces
  - KISS principle: "keep it simple, stupid!"
  - Break the problem down into small functions, each does only one simple task, and does it correctly
- Function allows programmer to focus on a function interface, hiding details of how it is implemented
- Reuse of code

Today's Outline

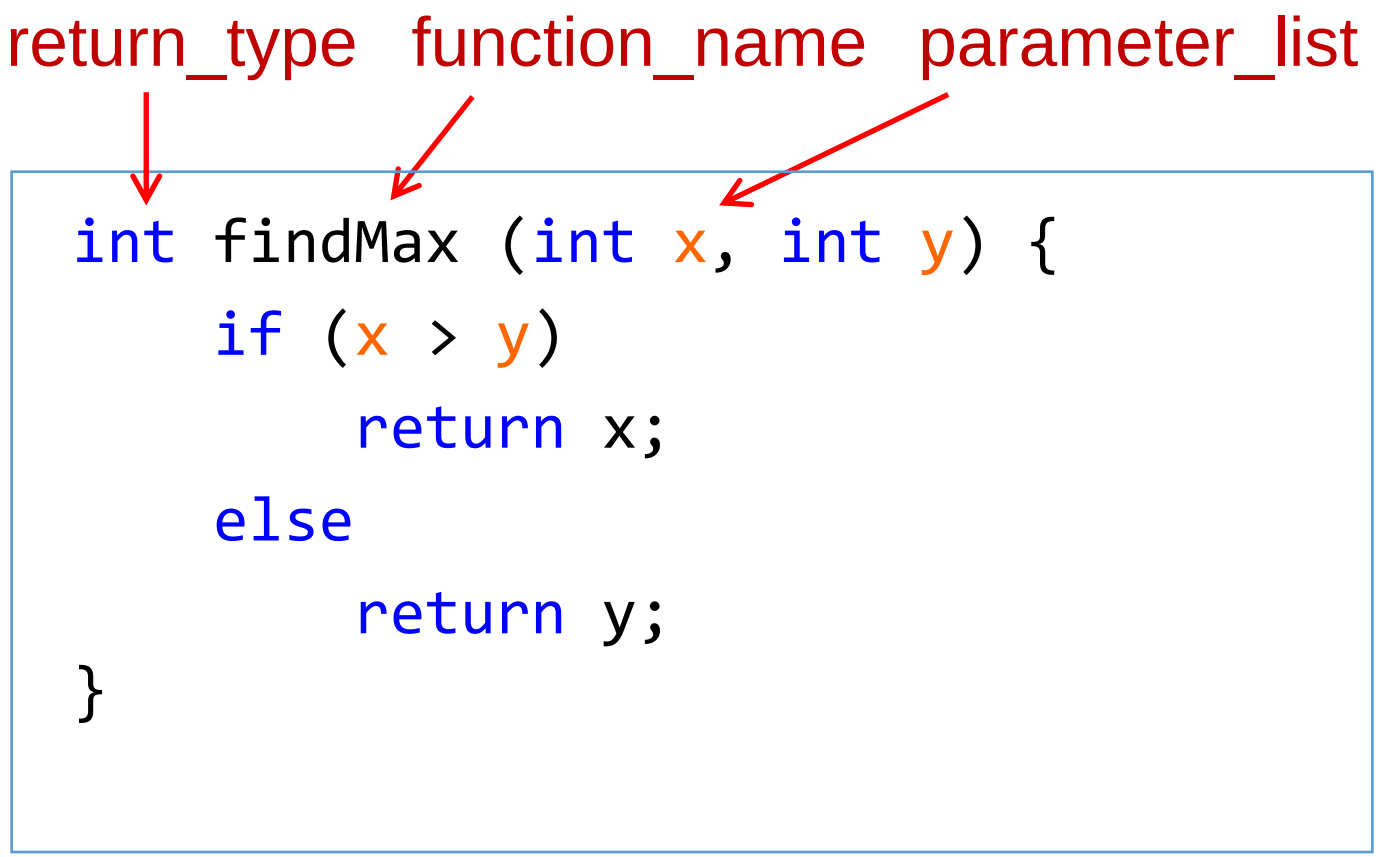
- Defining a function
- Calling a function
- Declare a function (function prototype)
- Function overload
- Passing parameters
- Recursive functions

Defining a Function



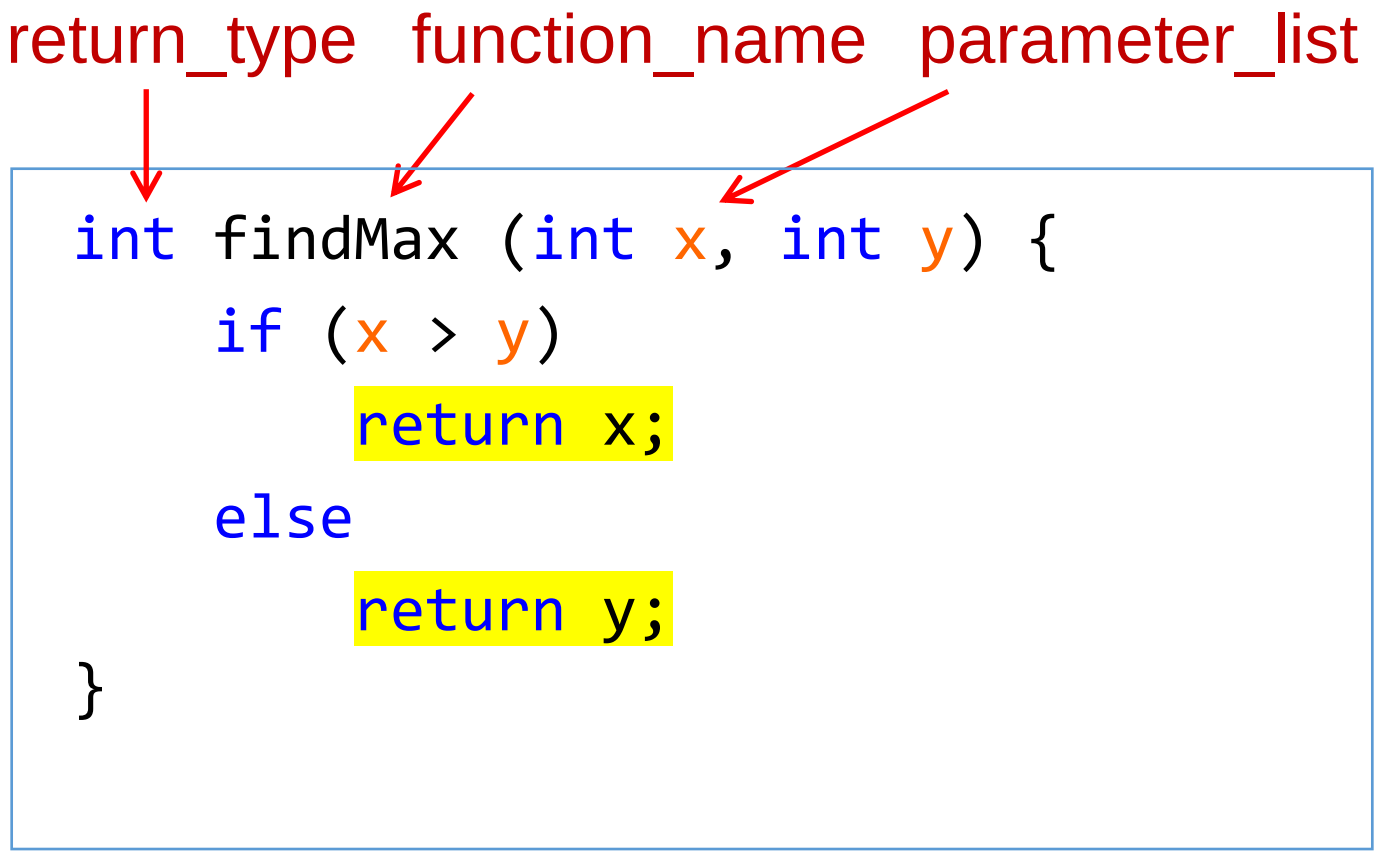
- return_type: A function may return some value.
  A return_type is the data type of the value the function returns. Some functions do not return any value. In this case, the return_type is void.
- function_name: The actual name of the function.
- parameter_list: The input arguments. The parameter list refers to the type and order of parameters. A function may contain no parameters.

Defining a Function



- return_type: A function may return some value.
  A return_type is the data type of the value the function returns. Some functions do not return any value. In this case, the return_type is void.
- function_name: The actual name of the function.
- parameter_list: The input arguments. The parameter list refers to the type and order of parameters. A function may contain no parameters.

Defining a Function



- return statement

  Syntax:

  return expression;

  return;

- when a return is encountered, the program will immediately go back to the calling function
- if expression exists, its value will be sent back to the calling function.
- if necessary, the returning value will be type converted to the type specified in the function definition

Today's Outline

- Defining a function
- Calling a function
- Declare a function (function prototype)
- Function overload
- Passing parameters
- Recursive functions

Calling a Function

- To make a function call, we only need to specify a function name and provide argument(s) in a pair of ( )

```
void main() {
  int x=4;
  printHello(x);
  cout << "bye";
}
```

```
void printHello(int n) {
  int i;
  for (i=0; i<n; i++)
     cout << "Hello\n";
}
```

Calling a Function

- when calling a function, no need to specify parameter and return type
  - e.g., the following code will cause syntax errors

```
void main() {
  int x=4;
  void printHello(x);
  cout << "bye";
}
```

```
void main() {
  int x=4;
  printHello(int x);
  cout << "bye";
}
```

```
void printHello(int n) {
  int i;
  for (i=0; i<n; i++)
     cout << "Hello\n";
}
```

Calling a Function (defined in library)

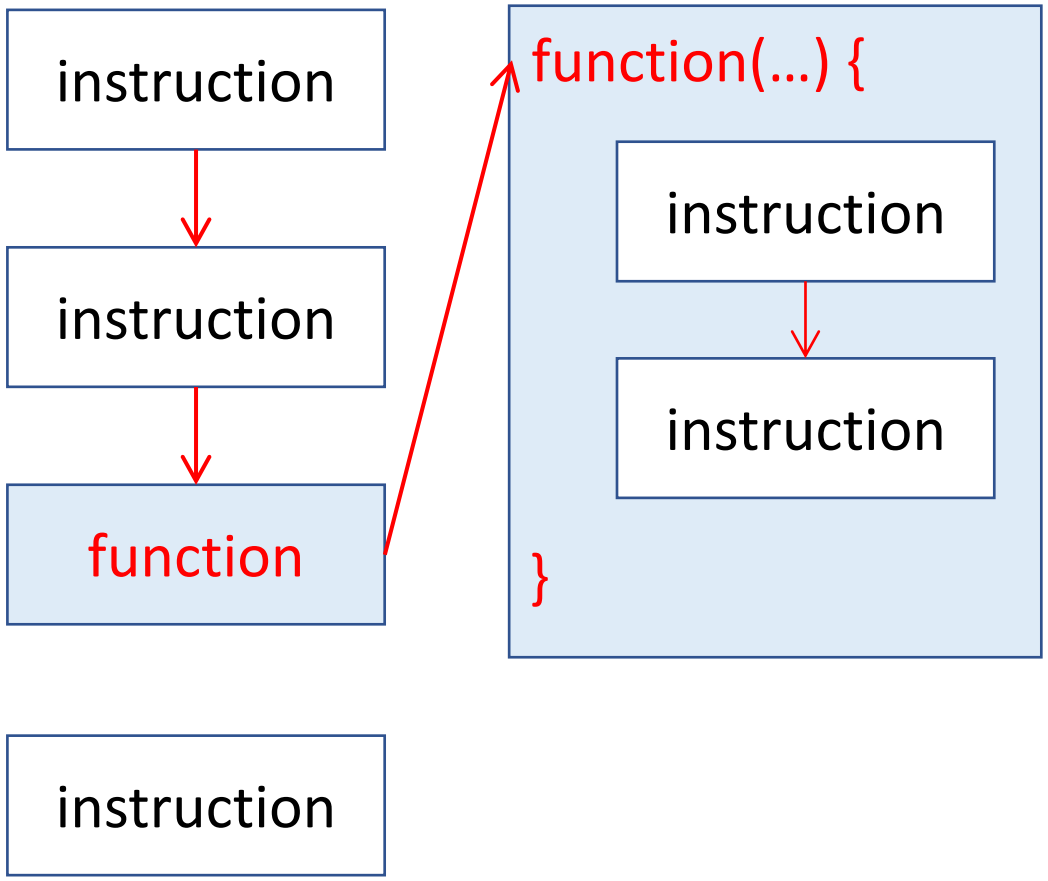
```
#include <iostream>     // tell the compiler that you are going to use functions
                        // defined in iostream library
using namespace std;
int main() {
    float area, side;
    cout << "Enter the area of a square: ";
    cin >> area;
    side = sqrt(area);  // pass area to the function sqrt which will return the
                        // square root of area
    cout << "The square has perimeter: " << 4*side;
    return 0;
}
```

Function in C++ Library

- The C++ standard library provides a rich collection of functions
- Mathematical calculations (#include <cmath>)
- String manipulations (#include <cstring>)
- Input/output (#include <iostream>)
- Some functions are defined in multiple library in some platform
  - e.g. function sqrt is defined in both cmath and iostream in VS

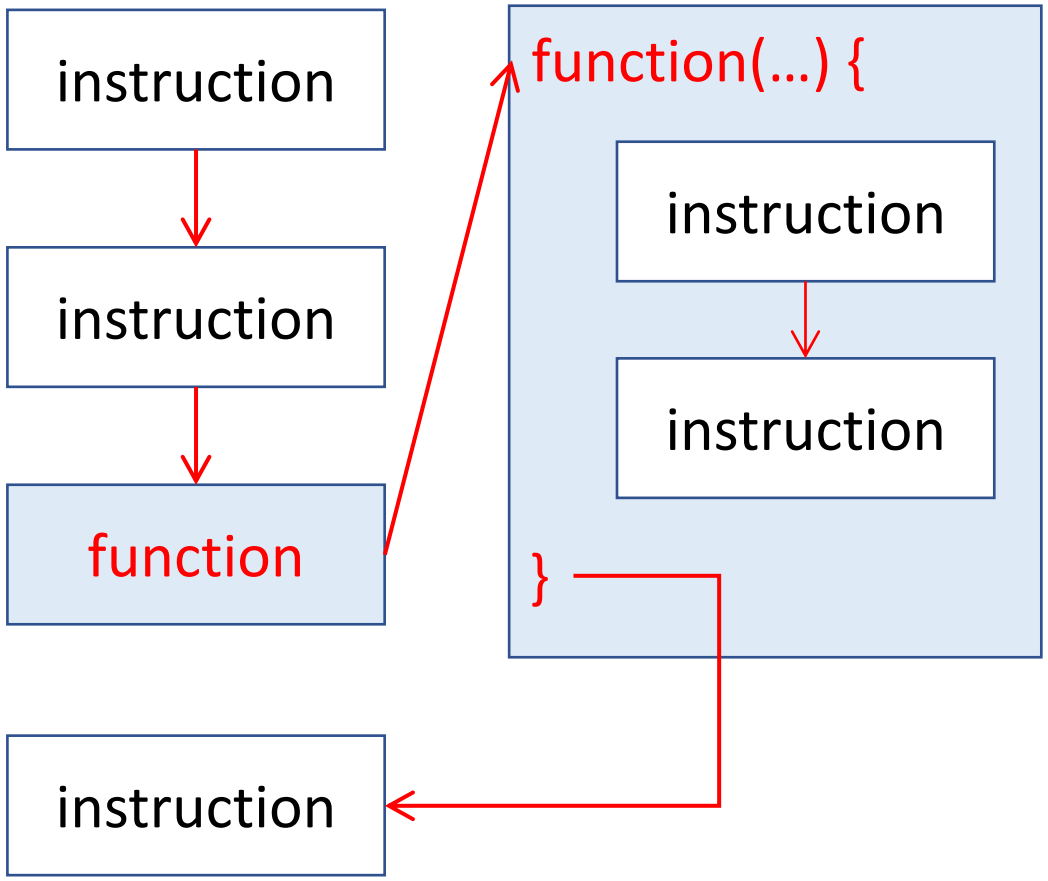
Flow of Function Call

- During program execution, when a function name followed by parentheses is encountered, the function is invoked and the program control is passed to that function;



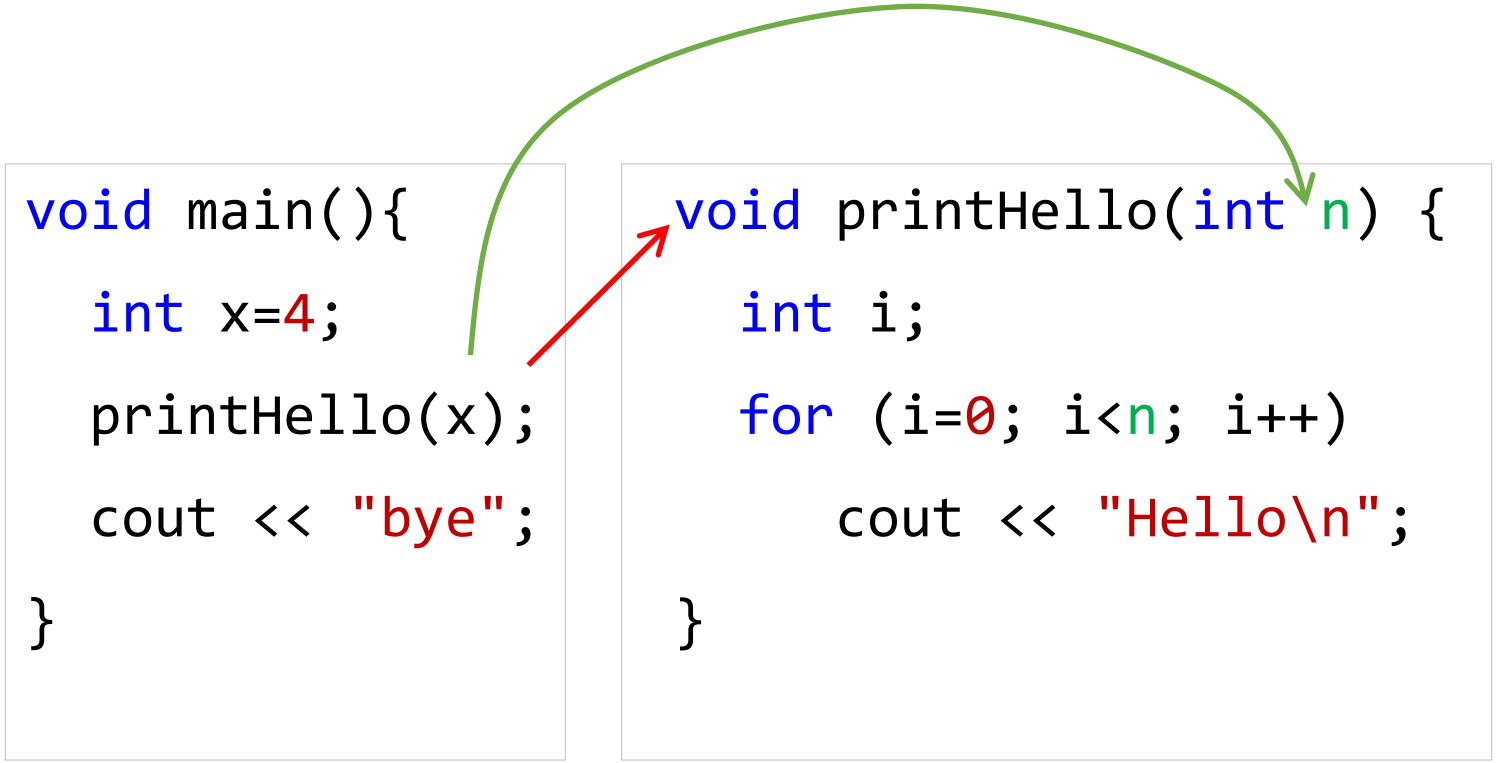
Flow of Function Call

- During program execution, when a function name followed by parentheses is encountered, the function is invoked and the program control is passed to that function;
- when the function ends, program control is returned to the statement immediately after the function call in the original function



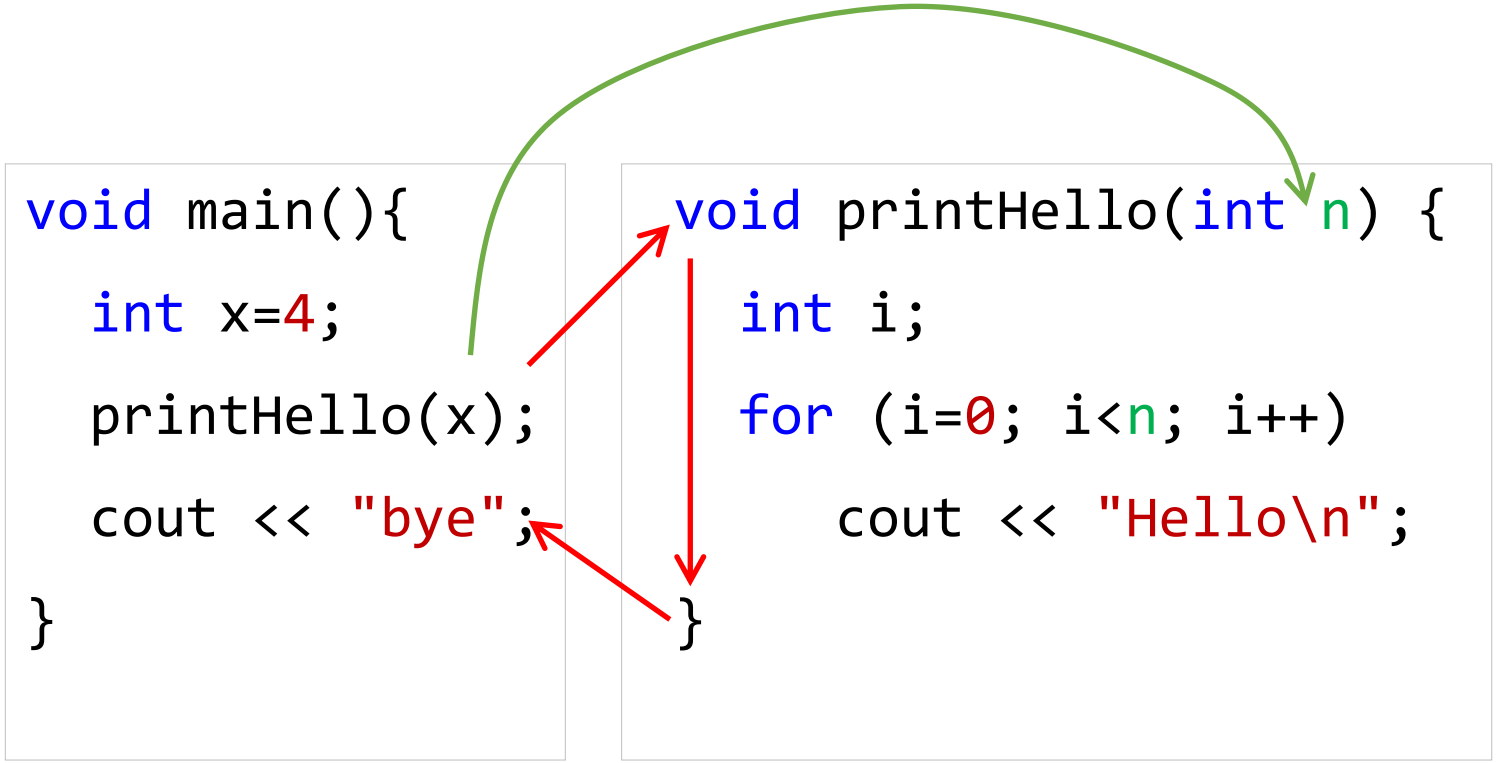
Flow of Function Call

1. program starts execution in main()
2. printHello is called
3. arguments passed to printHello



Flow of Function Call

1. program starts execution in main()
2. printHello is called
3. arguments passed to printHello
4. execute statements in printHello
5. control goes back to main() and "bye" is printed



Function in Memory Stack

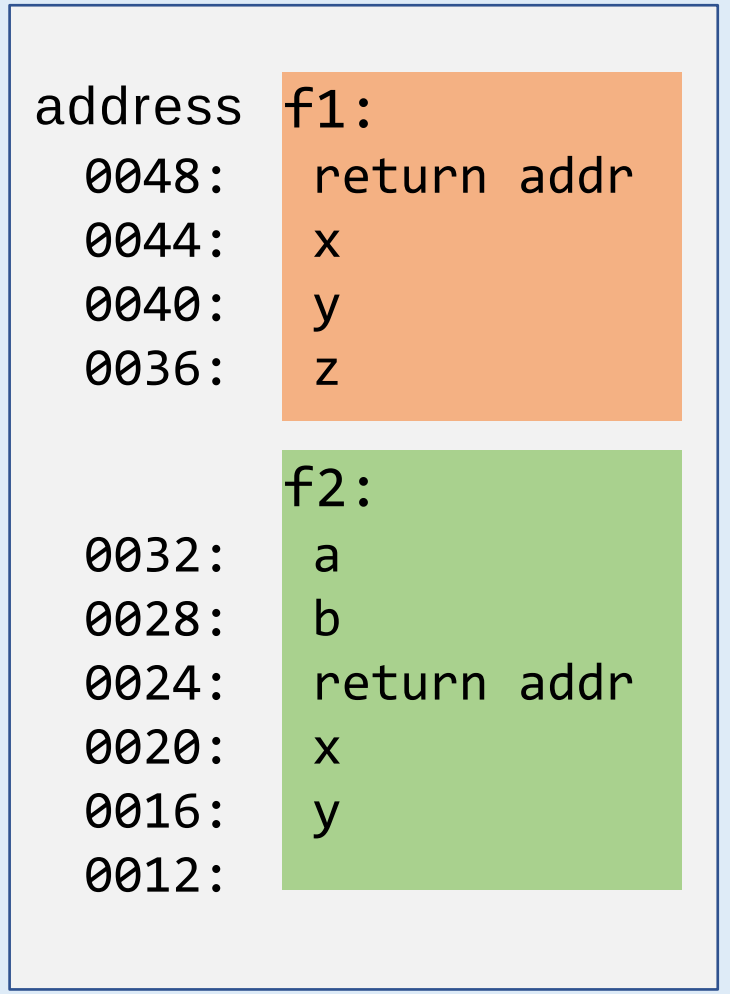
```
void f1() {
  int x = 5;
  int y = 6;
  int z = f2(x, y);
  cout << a << " " << z;
}
```

```
int f2(int a, int b) {
  int x = a + b;
  int y = b – a;
  a++;
  return x*y;
}
```

Each function has its own memory space which stores

- data (i.e., local variables and parameters)
- return address (address of the next instruction in the calling function)

Memory Stack



Today's Outline

- Defining a function
- Calling a function
- Declare a function (function prototype)
- Function overload
- Passing parameters
- Recursive functions

Function Prototype

- A function should be defined before use

```
// CORRECT
int findMax(int x, int y) {
   return (x > y) ? x:y;
}
void main() {
   cout << findMax(3, 4);
}
```

```
// SYNTAX ERROR
void main() {
   cout << findMax(3, 4); // findMax undefined
}
int findMax(int x, int y) {
   return x>y ? x:y;
}
```

Function Prototype

- A function should be defined before use

```
// CORRECT
int findMax(int x, int y) {
   return (x > y) ? x:y;
}
void main() {
   cout << findMax(3, 4);
}
```

```
// SYNTAX ERROR
void main() {
   cout << findMax(3, 4); // findMax undefined
}
int findMax(int x, int y) {
   return x>y ? x:y;
}
```

- Suppose we have 3 functions, where func1 calls func2, func2 calls func3, and func3 calls func1. In what order should the functions be defined?

Function Prototype

- A function should be defined before use

```
// CORRECT
int findMax(int x, int y) {
   return (x > y) ? x:y;
}
void main() {
   cout << findMax(3, 4);
}
```

```
// SYNTAX ERROR
void main() {
   cout << findMax(3, 4); // findMax undefined
}
int findMax(int x, int y) {
   return x>y ? x:y;
}
```

- Suppose we have 3 functions, where func1 calls func2, func2 calls func3, and func3 calls func1. In what order should the functions be defined?
- C++ allows us to bypass this problem using *function prototypes*

Function Prototype

- C++ allows us to *declare a function* and then call the function before defining it

- The declaration of the function is called *function prototype*, which
  - specifies the function name, parameters and return type
  - for example, the following statement declares foo as a function, there is no input and no return value

```
void foo (void);
```

Function Prototype

```
// SYNTAX ERROR
void main() {
    cout << findMax(3, 4); // findMax undefined
}

int findMax(int x, int y) {
    return x>y ? x:y;
}
```

Function Prototype

```
// SYNTAX ERROR
void main() {
   cout << findMax(3, 4); // findMax undefined
}

int findMax(int x, int y) {
   return x>y ? x:y;
}
```

```
// CORRECT
int findMax(int, int);

void main() {
   cout << findMax(3, 4);
}

int findMax(int x, int y) {
   return x>y ? x:y;
}
```

- int findMax(int, int) declares findMax as a function name, the return type is int, and there're two parameters and their types are int

Function Prototype

```
// SYNTAX ERROR
void main() {
    cout << findMax(3, 4); // findMax undefined
}

int findMax(int x, int y) {
    return x>y ? x:y;
}
```

```
// CORRECT
int findMax(int n1, int n2);

void main() {
    cout << findMax(3, 4);
}

int findMax(int x, int y) {
    return x>y ? x:y;
}
```

- Another way to declare the prototype is: int findMax(int n1, int n2);
- However, the variable names are optional, and you can use different parameter names in the actual function definition

Function Prototype

- void is used if a function takes no arguments
- Prototypes allow the compiler to check the code more thoroughly
- Arguments passed to function are coerced where necessary, e.g., printDouble(4) where the integer 4 will be *promoted* as a double type

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

void printDouble(double);

void main() {
    int x=4;
    printDouble(x);
}

void printDouble(double d) {
    cout << fixed;
    cout << d << endl;
}
```

Function Prototype

- In a large program, a function *f* may be used by many other functions written in different source files
  - where to declare function *f*?

Function Prototype

- In a large program, a function *f* may be used by many other functions written in different source files
  - where to declare function *f*?
- In C++, function prototype and definition can be stored separately
- Header file (.h):
  - With extension .h, e..g, stdio.h, string.h
  - Contain function prototype only
  - To be included in the program that will call the function
- Implementation file (.cpp)
  - Contain function implementation (definition)

Function Prototype

main.cpp

```
#include "mylib.h"

void main() {
   int x, y=2, z=3;
   x = calMin(y, z);
}
```

mylib.h

```
int calMin(int, int)
```

mylib.cpp

```
int calMin(int a, int b) {
   if (a>b)
      return b;
   else
      return a;
}
```

Today's Outline

- Defining a function
- Calling a function
- Declare a function (function prototype)
- Function overload
- Passing parameters
- Recursive functions

Function Overload

- *Overloading*: two or more functions with the *same name* but *different implementations*
- Two or more functions are said to be overloaded if they differ in
  - the number of arguments, OR
  - the type of arguments, OR
  - the order of arguments
- When an overloaded function is called, the compiler determines the most appropriate call by comparing function argument types

Overload: Example-I

```
void printData(double x) {
   cout << "Print double: " << x << endl;
}

void printData(float x) {
   cout << "Print float: " << x << endl;
}
```

```
int main() {
   double a = 0;
   float b = 0;
   printData(a);
   printData(b);
   return 0;
}
```

Overload: Example-II

```
double sum(double x, double y) {
   return x+y;
}

double sum(double x, double y, double z) {
   return x+y+z;
}
```

```
int main() {
   double a, b, c;
   cin >> a >> b >> c;
   cout << sum(a, b) << endl;
   cout << sum(a, b, c) << endl;
   return 0;
}
```

Overload: Common Errors

```
int sum(int x, int y) {
   return x+y;
}
int sum(int a, int b) {
   return a+b;
}
int main() {
   int a, b;
   cin >> a >> b;
   cout << sum(a, b);
   return 0;
}
```

```
int sum(int x, int y) {
   return x+y;
}
char sum(int x, int y) {
   return '0'+(char)(x+y);
}
int main() {
   int a, b;
   cin >> a >> b;
   char s = sum(a, b);
   cout << s;
   return 0;
}
```

Overload: Common Errors

- Ambiguous call: when the compiler is unable to choose between two correctly overloaded functions
- Automatic type conversions are the main cause of ambiguity

```
void printData(double x) {
   cout << "Print double: " << x << endl;
}
void printData(float x) {
   cout << "Print float: " << x << endl;
}
int main() {
   int a = 0;
   printData(a);
   return 0;
}
```

Overload: Common Errors

- Ambiguous call: when the compiler is unable to choose between two correctly overloaded functions
- Automatic type conversions are the main cause of ambiguity
- *Correction*: match the parameter type

```
void printData(double x) {
   cout << "Print double: " << x << endl;
}
void printData(float x) {
   cout << "Print float: " << x << endl;
}
int main() {
   int float a = 0;
   printData(a);
   return 0;
}
```

Today's Outline

- Defining a function
- Calling a function
- Declare a function (function prototype)
- Function overload
- Passing parameters
- Recursive functions

Argument vs Parameter

- Parameter: (a.k.a, *formal parameter*)
  - identifier that appears in function declaration
- Argument: (a.k.a, *actual parameter*)
  - expression that appears in a function call

```
void printHello(int n) {
  int i;
  for (i=0; i<n; i++)
     cout << "Hello\n";
}
```

```
void main() {
  int x=4;
  printHello(x);
  cout << "bye";
}
```

Parameter Passing in C++

- There're different ways in which arguments can be passed into the called function

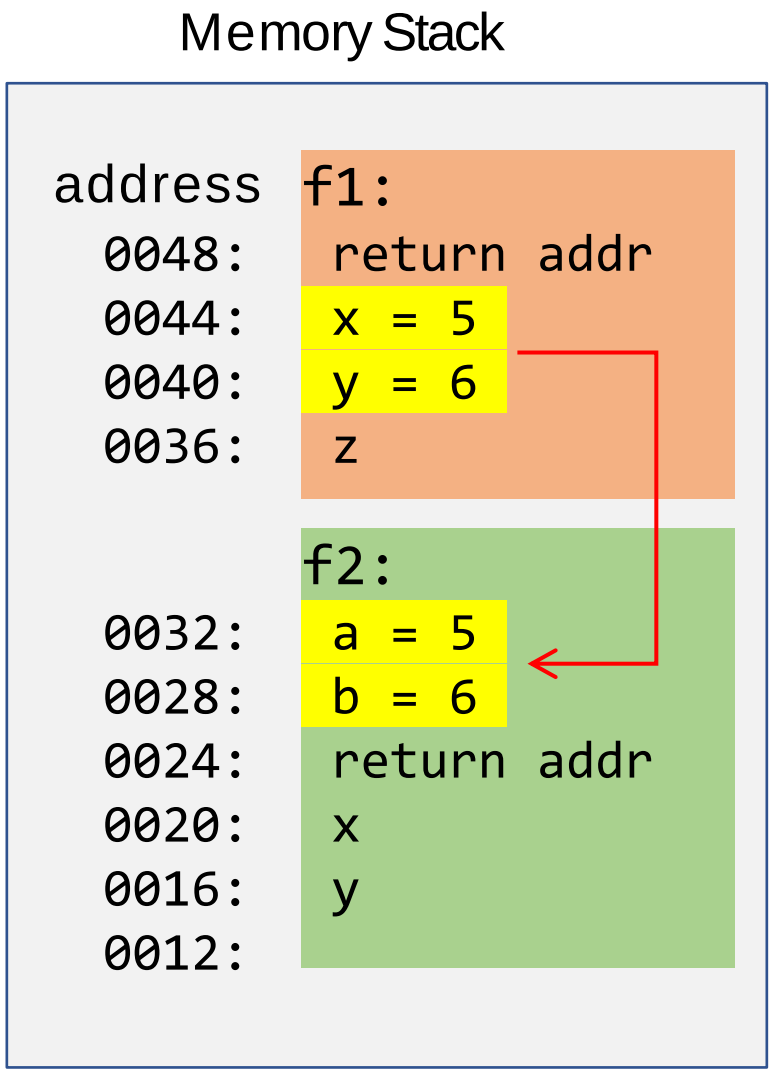
- Three most common methods
  - Pass-by-Value
  - Pass-by-Reference
  - Pass-by-Pointer (in later classes)

Pass-by-Value

- the value of argument is *copied* to parameter

```
void f1() {
  int x = 5;
  int y = 6;
  int z = f2(x, y);
  cout << a << " " << z;
}
```

```
int f2(int a, int b) {
  int x = a + b;
  int y = b - a;
  a++;
  return x*y;
}
```

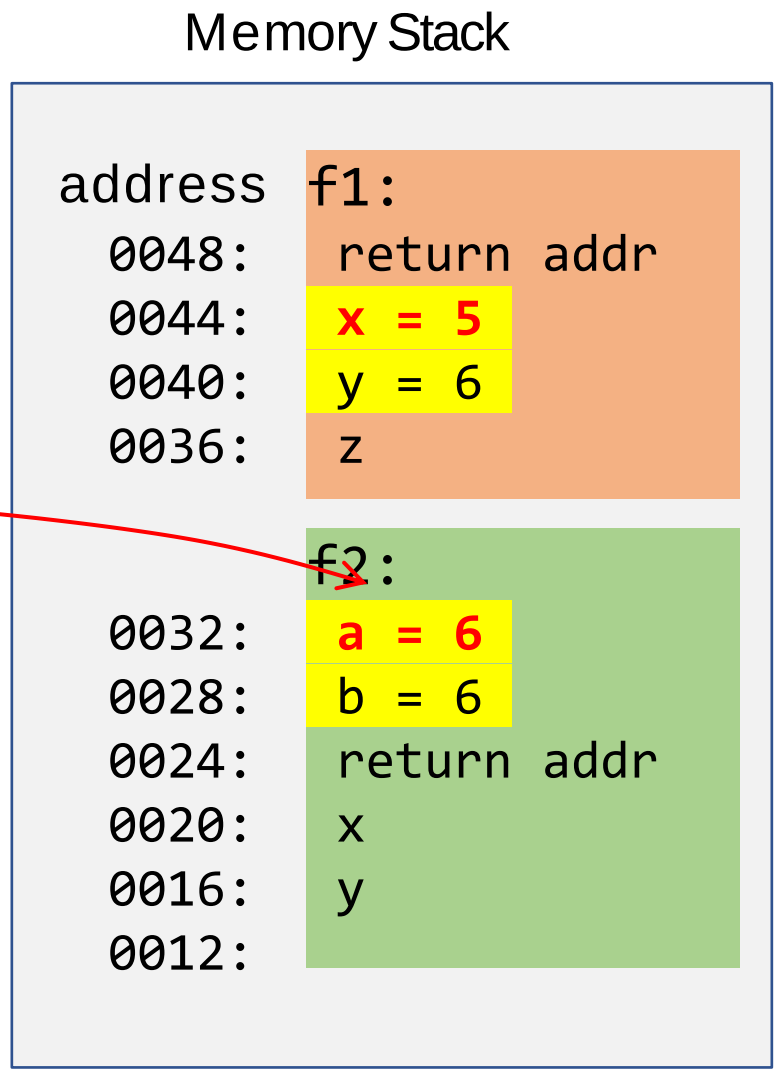


Pass-by-Value

- the value of argument is *copied* to parameter

```
void f1() {
  int x = 5;
  int y = 6;
  int z = f2(x, y);
  cout << a << " " << z;
}
```

```
int f2(int a, int b) {
  int x = a + b;
  int y = b - a;
  a++;
  return x*y;
}
```



- when a++ is executed in f2, only the memory storage of f2 is modified

Pass-by-Value

```
void func(int y) {
   y=4;
}

void main(){
   int y=3;
   func(y);
   cout << y << endl;
}
```

- y and y are two different variables stored in *different* places in memory
  - y (parameter) is a local variable in func
  - y (argument) is a local variable in main

Pass-by-Value

```
void func(int y) {
   y=4; // modify y in func(), not the one in main()
}

void main(){
   int y=3;
   func(y);
   cout << y << endl; // print 3, y remains unchanged
}
```

- y and y are two different variables stored in *different* places in memory
  - y (parameter) is a local variable in func
  - y (argument) is a local variable in main
- => Modifying y in func doesn't affect y

Pass-by-Value

```
void int func(int y) {
    y=4; // modify the value of x to 4
    return y;
}

void main(){
    int y=3;
    func(y); y=func();
    cout << y << endl; // print 4
}
```

- How to modify y in func()?
- By assigning the return value of f(y) to y
- After function call, y gets a value of 4

Pass-by-Value: Exercises

- What's the output of the following program?

```
void f(int y, int x) {
   cout << "x=" << x << endl;
   cout << "y=" << y << endl;
}

void main(){
   int x=3, y=4;
   f(x, y);
}
```

Pass-by-Value: Exercises

- Finding the max of 3 numbers, i, j, k

```
void main() {
   int i,j,k;
   int max;
   cin >> i >> j >> k;

   // find the max of i, j, k
   ___ = findMax(___, ___);
   ___ = findMax(___, ___);

   cout << "max is " << max;
}
```

```
int findMax(int n1, int n2) {
   if (n1>n2)
      return n1;
   else
      return n2;
}
```

Pass-by-Value: Exercises

- Finding the max of 3 numbers, i, j, k

```
void main() {
   int i,j,k;
   int max;
   cin >> i >> j >> k;

   // find the max of i, j, k
   max = findMax(i, j);
   max = findMax(k, max);

   cout << "max is " << max;
}
```

```
int findMax(int n1, int n2) {
   if (n1>n2)
      return n1;
   else
      return n2;
}
```

Pass-by-Reference

- Argument *address* is passed to the parameter
- *Argument can be updated inside the function*
- Add '&' in front of the parameter that to be pass by reference

```
void swap(int &a, int &b) {
    int tmp = a;
    a = b;
    b = tmp;
}
```

```
void main() {
    int x=1, y=3;
    swap(x, y);
    cout << "x:" << x << ", y:" << y << endl;
}
```

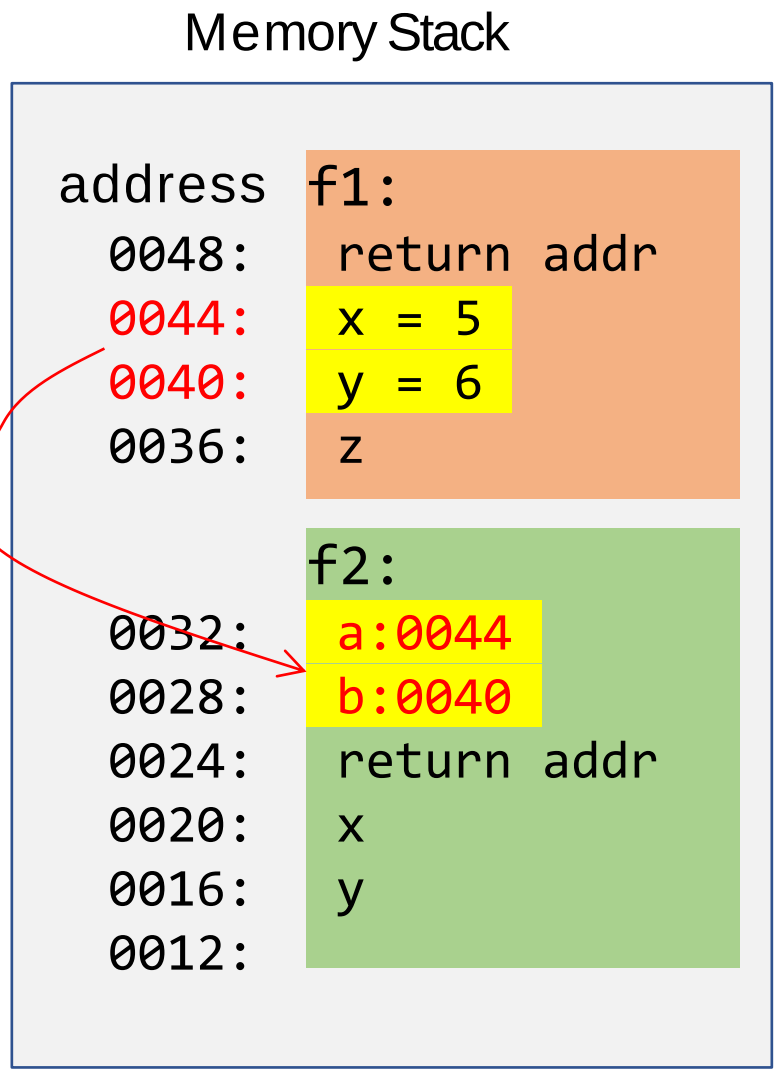
- More details will be explained in future lecture (pointer)

Pass-by-Reference

```
void f1() {
  int x = 5;
  int y = 6;
  int z = f2(x, y);
  cout << a << " " << z;
}
```

```
int f2(int &a, int &b) {
  int x = a + b;
  int y = b - a;
  a++;
  return x*y;
}
```

- the address of x (0044) is passed to f2



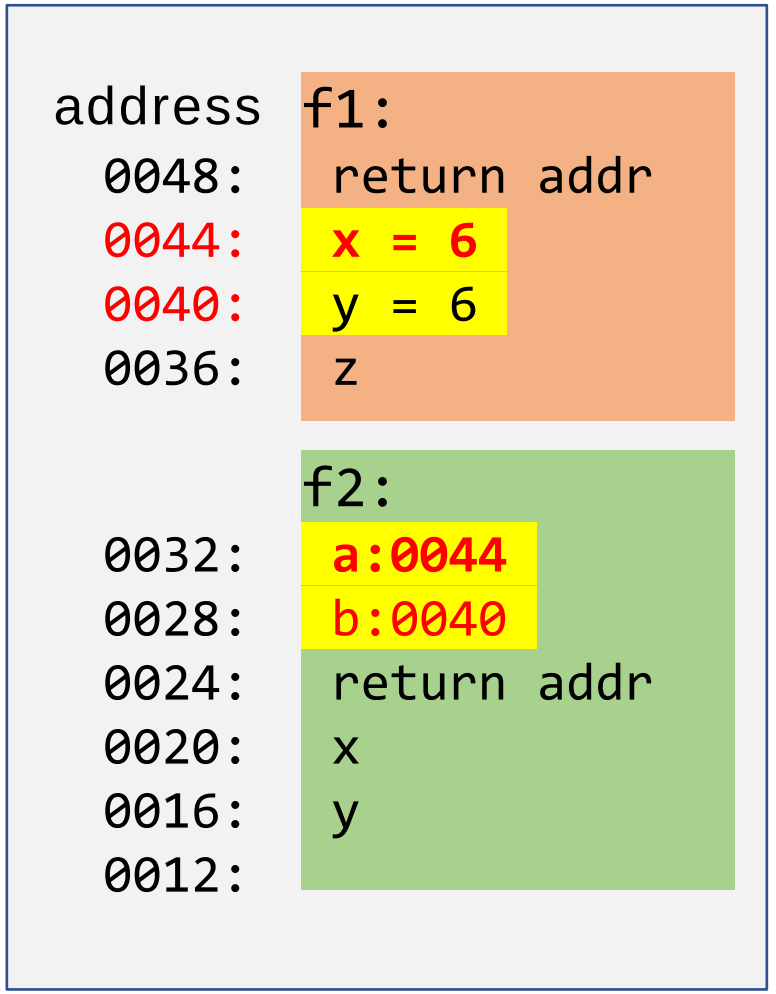
Pass-by-Reference

```
void f1() {
  int x = 5;
  int y = 6;
  int z = f2(x, y);
  cout << a << " " << z;
}
```

```
int f2(int &a, int &b) {
  int x = a + b;
  int y = b - a;
  a++;
  return x*y;
}
```

- the address of x (0044) is passed to f2
- when a++ is executed in f2, the value stored in 0044 (i.e., x in f1) is modified

Memory Stack



Parameter Passing: Default Parameters

- We can provide some default values for certain parameters
- Example

```
void f(int a, int b=0) { // parameter b with a default value 0
   cout << a << " " << b << endl;
}
void main() {
   f(1, 2);
   f(3); // you can call f without providing argument for b, in this case,
         // the default value 0 will be used for b in f
}
```

Parameter Passing: Default Parameters

- All the default parameters MUST locate at the right side of normal parameters
- Invalid example

```
void f(int a, int b=0, int c) { // invalid definition, default parameter b
                                // located at left side of normal parameter c
   cout << a << " " << b << endl;
}
void main() {
   f(1, 2);
   f(3);
}
```

Today's Outline

- Defining a function
- Calling a function
- Declare a function (function prototype)
- Passing parameters
- Recursive functions

Recursions

- One basic problem solving technique is to break the task into subtasks
- If a subtask is a smaller version of the original task, you can solve the original task using a recursive function
- A recursive function is one that invokes itself, either directly or indirectly

Example: Factorial

- The factorial of n is defined as

```
0! = 1
n! = n*(n-1)*… 2*1, for n > 0
```

- A recurrence relation: (induction)

```
n! = n*(n-1)!,        for n > 0
```

- e.g.,

```
3! = 3 * 2
   = 3 * 2 * 1!
   = 3 * 2 * 1 * 0!
   = 3 * 2 * 1 * 1
```

Iterative vs Recursive

Iterative

```
int factorial(int n) {
    int i, fact=1;
    for (i=1; i<=n; i++) {
        fact = i*fact;
    }
    return fact;
}
```

Iterative vs Recursive

Iterative

```
int factorial(int n) {
    int i, fact=1;
    for (i=1; i<=n; i++) {
        fact = i*fact;
    }
    return fact;
}
```

Recursive

```
int factorial(int n) {
    if (n==0)
        return 1;
    return n*factorial(n-1);
}
```

Example: Vertical Number

- Input: one (non-negative) integer
- Output: integer with one digit per line, from left to right

Input	Output
12345	1 2 3 4 5
7894	7 8 9 4
4	4

Example: Vertical Number

- An *iterative* logic flow: use a *for* loop to print each digit, from left to right
  - First, you need to know how many digits does the number have
    - i.e., how many rounds does your loop need to iterate
  - Second, you must be able to get a digit at a specific position
    - i.e., to print the i-th digit from the right in the i-th loop iteration

Input	Output
12345	1 2 3 4 5
7894	7 8 9 4
4	4

Example: Vertical Number

- An *iterative* logic flow: use a *for* loop to print each digit, from left to right
  - First, you need to know how many digits does the number have
    - i.e., how many rounds does your loop need to iterate
  - Second, you must be able to get a digit at a specific position
    - i.e., to print the i-th digit from the right in the i-th loop iteration
- *Let's develop two functions*

Input	Output
12345	1 2 3 4 5
7894	7 8 9 4
4	4

Example: Vertical Number

- How to count the number of digits in a number?

```
int countDigits(int num) {
        int n = 1;

        while ((num / pow(10, n-1)) != 0) // pow(x, y) return the result of x^y as a float
                n++;

        return n;
}
```

Example: Vertical Number

- How to count the number of digits in a number?

```
// Any problems in this code ??

int countDigits(int num) {
        int n = 1;

        while ((num / pow(10, n-1)) != 0) // pow(x, y) return the result of x^y as a float
                n++;

        return n;
}
```

Example: Vertical Number

- How to count the number of digits in a number?

```
// Any problems in this code ??

int countDigits(int num) {
        int n = 1;


        while ((num / pow(10, n-1)) != 0) // pow(x, y) return the result of x^y as a float
                n++;
        // To consider the special case when num == 0
        return max(n, 1);
}
```

Example: Vertical Number

- How to count the number of digits in a number?

```
// Any problems in this code ??

int countDigits(int num) {
        int n = 1;
        // Watch out implicit type conversion
        while ((int)(num / pow(10, n-1)) != 0) // pow(x, y) return the result of x^y as a float
                n++;
        // To consider the special case when num == 0
        return max(n, 1);
}
```

Example: Vertical Number

- How to get the *i*-th rightmost digits of a number?

```
int getDigit(int num, int i) {
        // Move num to the right by i-1 digit(s)
        int tmp = num / pow(10, i-1);
        // Return the rightmost digit
        return tmp % 10;
}
```

Example: Vertical Number

- Put functions together

```
int main() {
        int num;
        cout << "Please enter a non-negative integer: ";
        cin >> num;
        int numOfDigits = countDigits(num);
        for (int i = 0; i < numOfDigits; i++)
                cout << getDigit(num, numOfDigits-i) << endl;
}
```

Input	Output
12345	1 2 3 4 5
7894	7 8 9 4
4	4

Example: Vertical Number

- A *recursive* logic flow:

Given a non-negative integer of *n* digits

$$d_n\ d_{n-1}\ \ldots\ d_1$$

- STEP-1: Solve a smaller version of the problem
  - i.e., print vertical number for $d_n\ d_{n-1}\ \ldots\ d_2$
- STEP-2: Print d_1

Input	Output
12345	1 2 3 4 5
7894	7 8 9 4
4	4

Example: Vertical Number

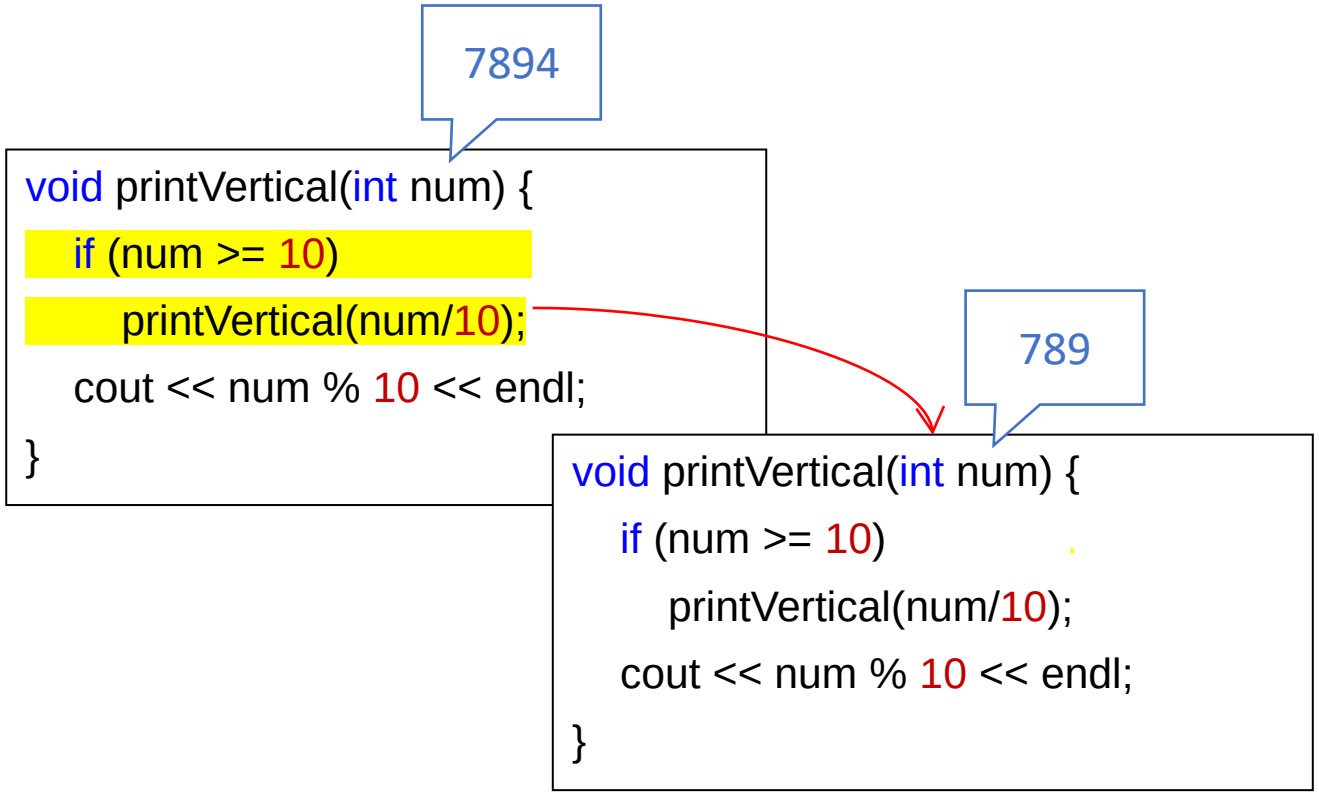
```
int main() {
        int num;
        cout << "Please enter a non-negative integer: ";
        cin >> num;
        printVertical(num);
}
```

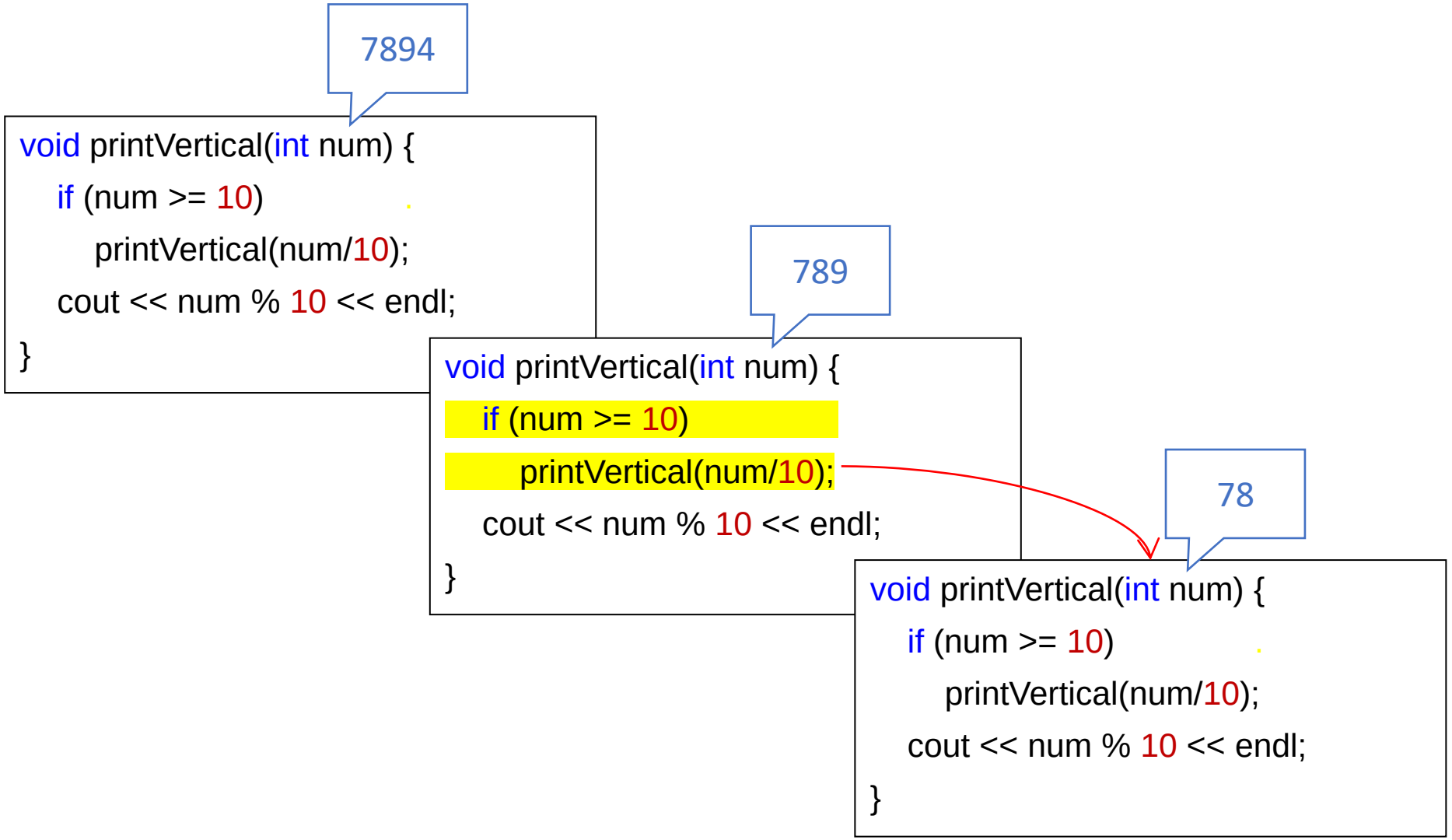
Input	Output
12345	1 2 3 4 5
7894	7 8 9 4
4	4

Example: Vertical Number

```
void printVertical(int num) {
        if (num >= 10) // has more than one digits
                printVertical(num/10);
        cout << num % 10 << endl;
}
int main() {
        int num;
        cout << "Please enter a non-negative integer: ";
        cin >> num;
        printVertical(num);
}
```

Input	Output
12345	1 2 3 4 5
7894	7 8 9 4
4	4


7894
void printVertical(int num) {
if (num >= 10)
printVertical(num/10);
cout << num % 10 << endl;
}
789
void printVertical(int num) {
if (num >= 10)
printVertical(num/10);
cout << num % 10 << endl;
}


7894
void printVertical(int num) {
if (num >= 10)
printVertical(num/10);
cout << num % 10 << endl;
}
789
void printVertical(int num) {
if (num >= 10)
printVertical(num/10);
cout << num % 10 << endl;
}
78
void printVertical(int num) {
if (num >= 10)
printVertical(num/10);
cout << num % 10 << endl;
}

7894

```
void printVertical(int num) {
    if (num >= 10)
        printVertical(num/10);
    cout << num % 10 << endl;
}
```

789

```
void printVertical(int num) {
    if (num >= 10)
        printVertical(num/10);
    cout << num % 10 << endl;
}
```

78

```
void printVertical(int num) {
    if (num >= 10)
        printVertical(num/10);
    cout << num % 10 << endl;
}
```

7

```
void printVertical(int num) {
    if (num >= 10)
        printVertical(num/10);
    cout << num % 10 << endl;
}
```

7894

```
void printVertical(int num) {
    if (num >= 10)
        printVertical(num/10);
    cout << num % 10 << endl;
}
```

789

```
void printVertical(int num) {
    if (num >= 10)
        printVertical(num/10);
    cout << num % 10 << endl;
}
```

78

```
void printVertical(int num) {
    if (num >= 10)
        printVertical(num/10);
    cout << num % 10 << endl;
}
```

7

```
void printVertical(int num) {
    if (num >= 10)
        printVertical(num/10);
    cout << num % 10 << endl;
}
```

7

7894

```
void printVertical(int num) {
    if (num >= 10)
        printVertical(num/10);
    cout << num % 10 << endl;
}
```

789

```
void printVertical(int num) {
    if (num >= 10)
        printVertical(num/10);
    cout << num % 10 << endl;
}
```

78

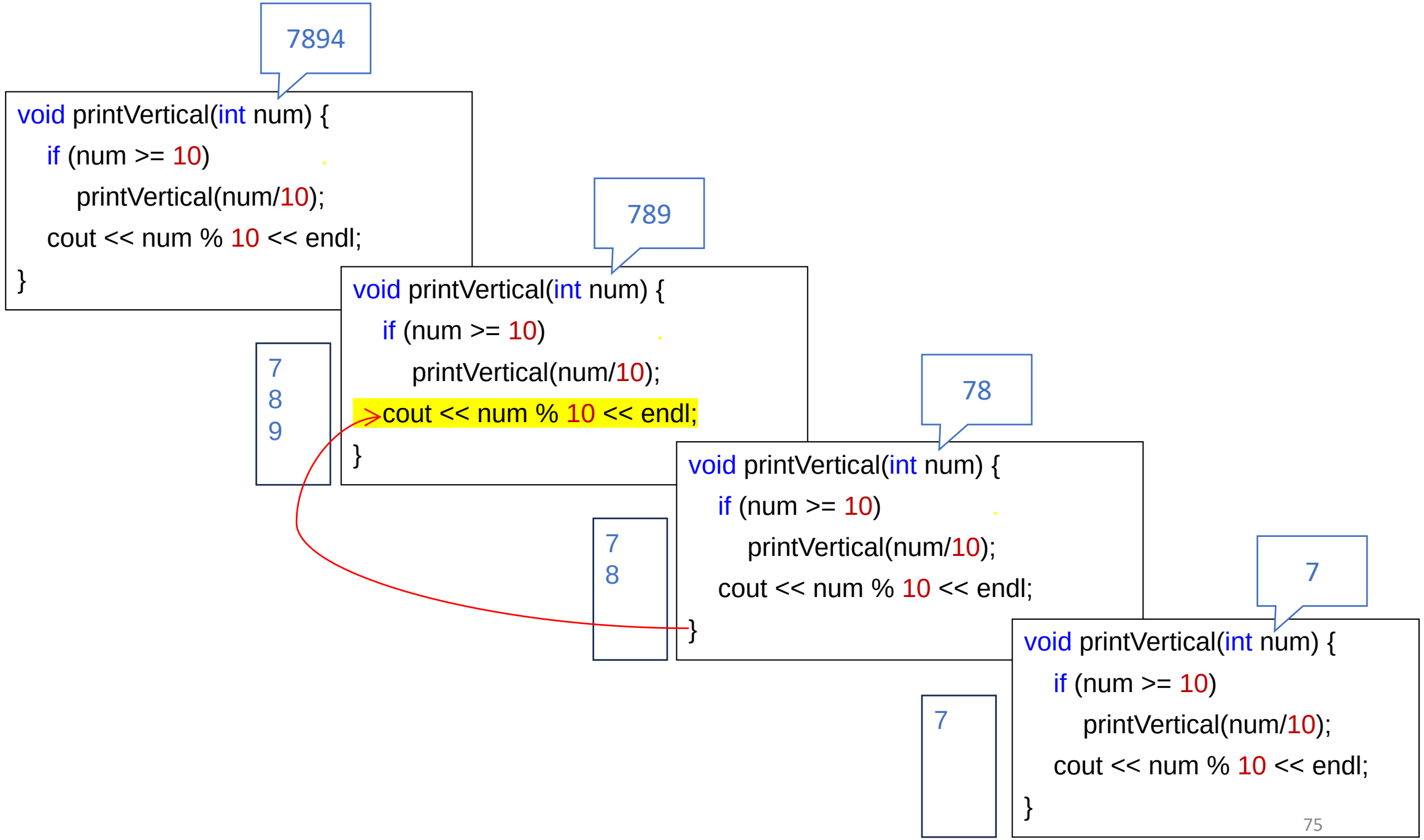
```
void printVertical(int num) {
    if (num >= 10)
        printVertical(num/10);
    cout << num % 10 << endl;
}
```

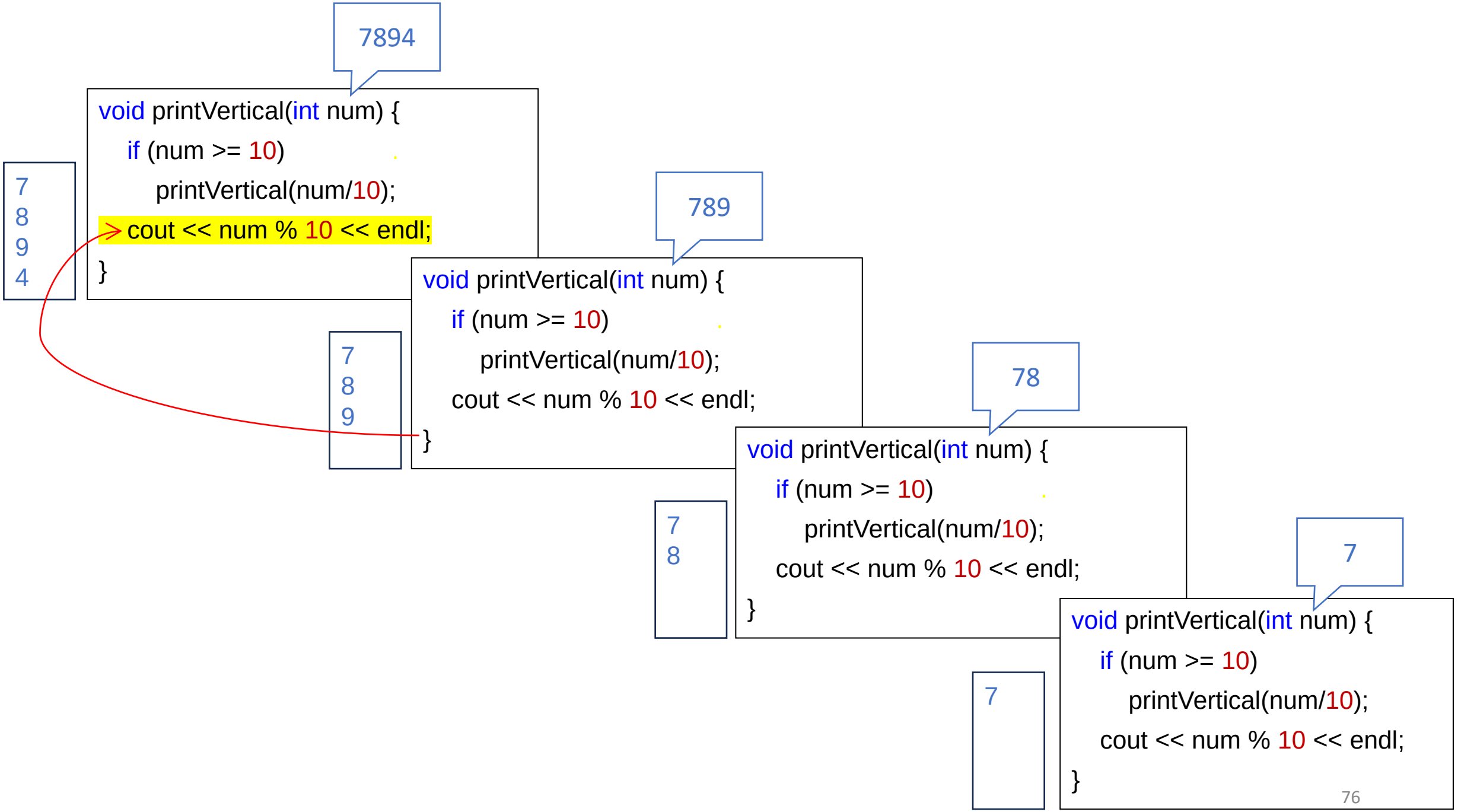
7
8

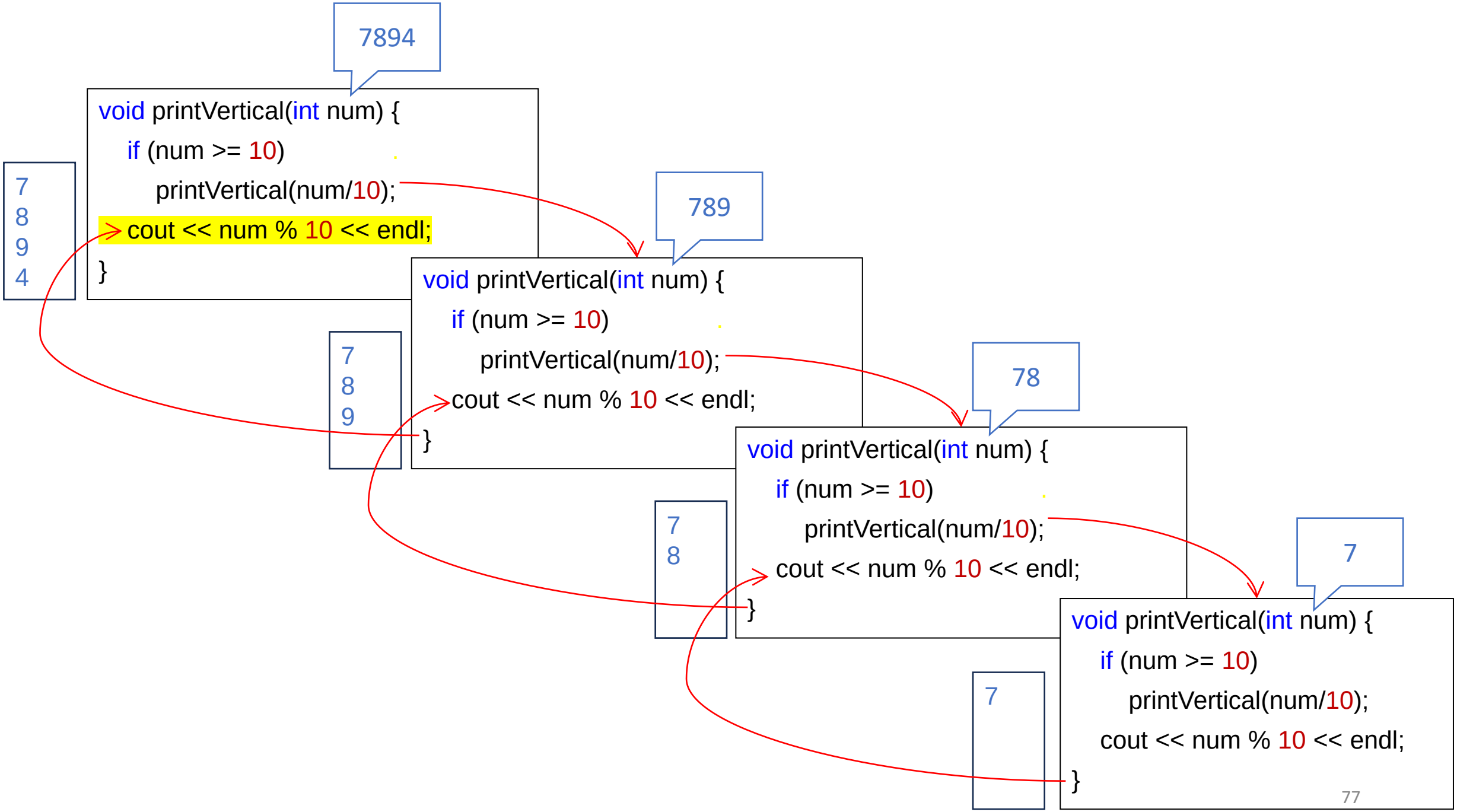
7

```
void printVertical(int num) {
    if (num >= 10)
        printVertical(num/10);
    cout << num % 10 << endl;
}
```

7


7894
void printVertical(int num) {
if (num >= 10)
printVertical(num/10);
cout << num % 10 << endl;
}
789
7
8
9
void printVertical(int num) {
if (num >= 10)
printVertical(num/10);
cout << num % 10 << endl;
}
78
7
8
void printVertical(int num) {
if (num >= 10)
printVertical(num/10);
cout << num % 10 << endl;
}
7
7
void printVertical(int num) {
if (num >= 10)
printVertical(num/10);
cout << num % 10 << endl;
}


7894
void printVertical(int num) {
if (num >= 10)
printVertical(num/10);
cout << num % 10 << endl;
}
7
8
9
4
789
void printVertical(int num) {
if (num >= 10)
printVertical(num/10);
cout << num % 10 << endl;
}
7
8
9
78
void printVertical(int num) {
if (num >= 10)
printVertical(num/10);
cout << num % 10 << endl;
}
7
8
7
void printVertical(int num) {
if (num >= 10)
printVertical(num/10);
cout << num % 10 << endl;
}
7


7894
void printVertical(int num) {
if (num >= 10)
printVertical(num/10);
cout << num % 10 << endl;
}
7
8
9
4
789
void printVertical(int num) {
if (num >= 10)
printVertical(num/10);
cout << num % 10 << endl;
}
7
8
9
78
void printVertical(int num) {
if (num >= 10)
printVertical(num/10);
cout << num % 10 << endl;
}
7
8
7
void printVertical(int num) {
if (num >= 10)
printVertical(num/10);
cout << num % 10 << endl;
}
7

Guidelines for Recursive Functions

- Identify the parameters
  - e.g., *n* in the factorial, *num* in printVertical
- Find out a recurrence relation between the current problem and a smaller version of itself (in terms of smaller parameters)
  - e.g., factorial(n) = n*factorial(n-1)
- Find out the base cases and their solutions
  - e.g., for factorial *n*=0; for printVertical, *num* has only one digit
  - Omitting the base case is one of the common mistakes in writing recursive functions

Checkpoints

1. There is no infinite recursion (check exist condition)

2. The break down of the problem works correctly

3. For each of cases that involve recursion, if all recursive calls perform their actions correctly, then the entire case performs correctly.

Checkpoints

	Factorial	Vertical Number
Exit condition	n == 0	n < 10
Problem break down	factorial(n) = n*factorial(n-1) e.g. n=2 --> 2!=2*1! n=3 --> 3!=3*2! n=4 --> 4!=4*3!	printDigits(n/10); cout << n%10; e.g. n=78 --> 7 was printed n=789 --> 7, 8 were printed n=7894 --> 7, 8, 9 were printed
If all stopping case are correct	n! is returned	n digits are printed

Efficiency of Recursion

- Generally speaking, non-recursive versions will execute more efficiently (time/space)
  - Overhead involved in entering and exiting blocks is avoided in non-recursive solutions.
  - Also have a number of local variables and temporaries that do not have to be saved and restored via a stack.
- There are conflicts between
  - Machine efficiency and
  - Programmer efficiency

Exercise-I: Fibonacci Number

- The Fibonacci numbers, commonly denoted F(n), form a sequence, called the Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from 0 and 1. That is,

$$F(0) = 0, F(1) = 1$$

$$F(n) = F(n-1) + F(n-2), \text{ for } n > 1$$

- Given n, calculate F(n).

Example:

- Input: n = 4
- Output: F(4) = 3

Exercise-II: Guessing Number

- Write a program to guess the number in the user's mind.

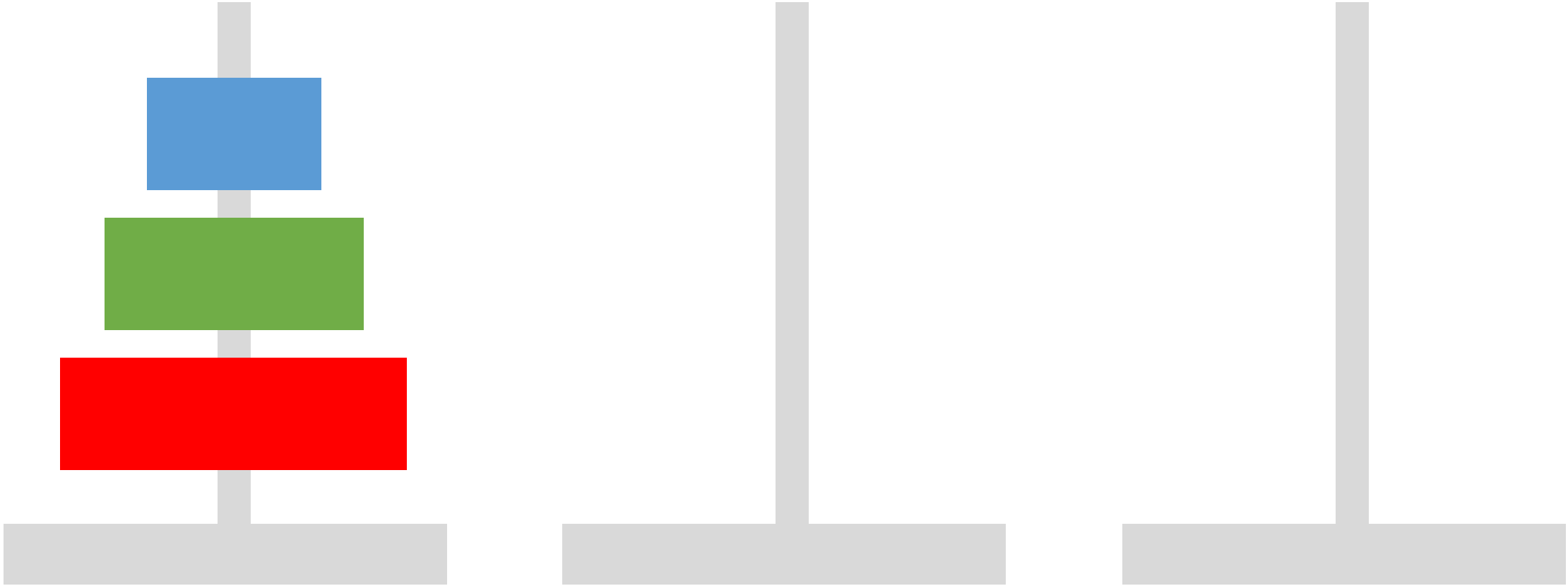
- Each round of guessing, the program should guess a number and ask the user if it is the correct number. If it is, the user should type 'Y'. Otherwise, the user should type '>' or '<' to indicate whether the guessed number is greater or smaller than the correct number.

- Solve the problem with recursion

Exercise-III: Tower of Hanoi

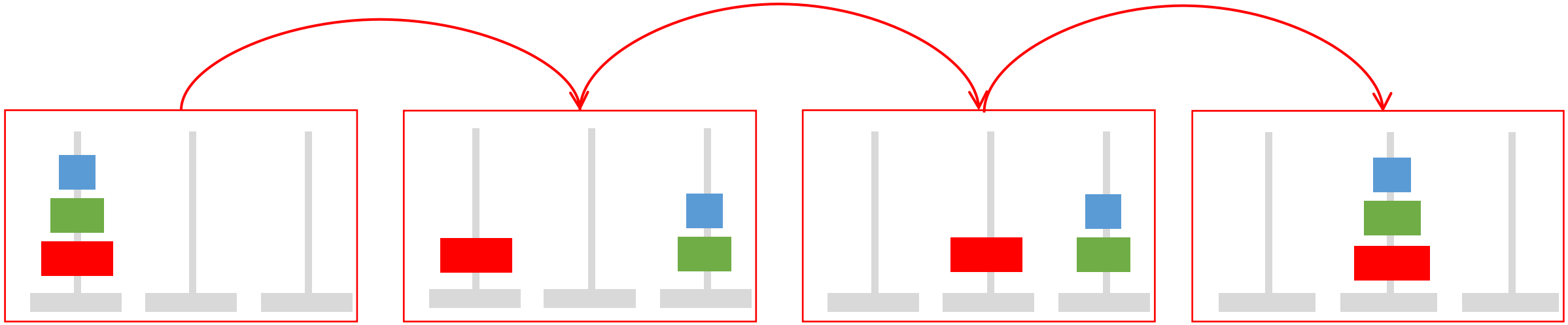
- 3 towers and n disks
- the smallest disk is on top and the largest is on the bottom
- you need to move all of the disks from the first tower to the second tower
- the third tower can be used to temporarily hold disks
- only one disk can be moved at a time.
- NO larger disk is allowed atop a smaller disk

Exercise-III: Tower of Hanoi



Exercise-III: Tower of Hanoi

- Decompose the problem



Exercise-III: Tower of Hanoi

```
// a:from tower, b:to tower, c: auxiliary tower
// n: number of disks to move

void Hanoi(int n, char a, char b, char c) {
   if (n == 1)                  // base case
      Move(a, b);
   else {                       // recursion
      Hanoi(n-1, a, c, b);  // Move top n-1 disks from A to C using B as auxiliary
      Move(a, b);              // Move the last disk from A to B
      Hanoi(n-1, c, b, a);  // Move the n-1 disks on C to B using A as auxiliary
   }
}

void Move(char a, char b) {
   cout << "Move from " << a << " to " << b << endl;
}
```

Summary

- Define, call, and declare functions
- Parameter passing
  - by value
  - By reference
- Recursive functions