

CS2311 Computer Programming

LT05: Function

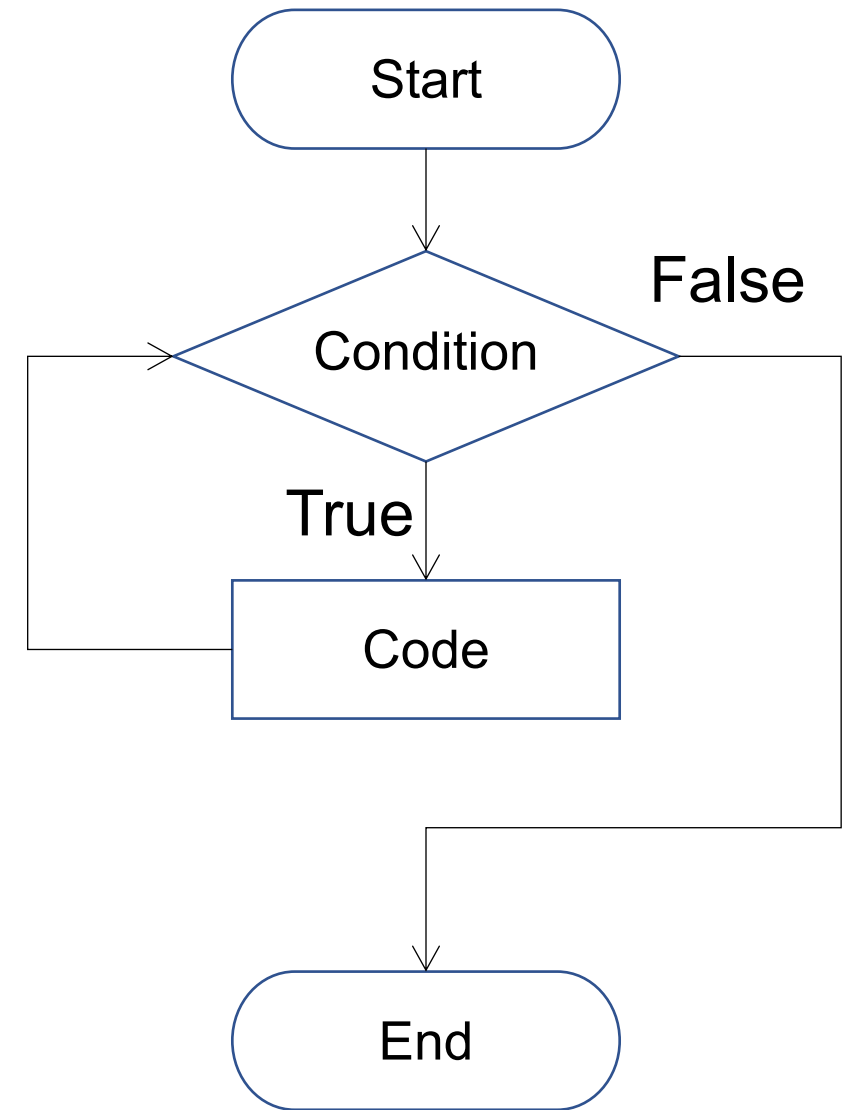
Computer Science, City University of Hong Kong
Semester B 2022-23

About Midterm

- Week 8 lecture time (Mar 10 Friday)
- In classroom, on paper (written based)
 - Formal proof needed for request of absence
- 90 minutes
- From Lec 1 (Intro) to Lec 6 (Array)
- 15% of final mark
- Sample questions and announcement will be released next week

Quick Review: Loop

- When the execution enters a **loop**, it executes a block of code **repeatedly** as long as a loop **condition** is met
 - Loop body
 - Iteration
- Beside sequential and branch execution loop is another common control flow
- while / do-while / for



Quick Review: Loop (cont'd)

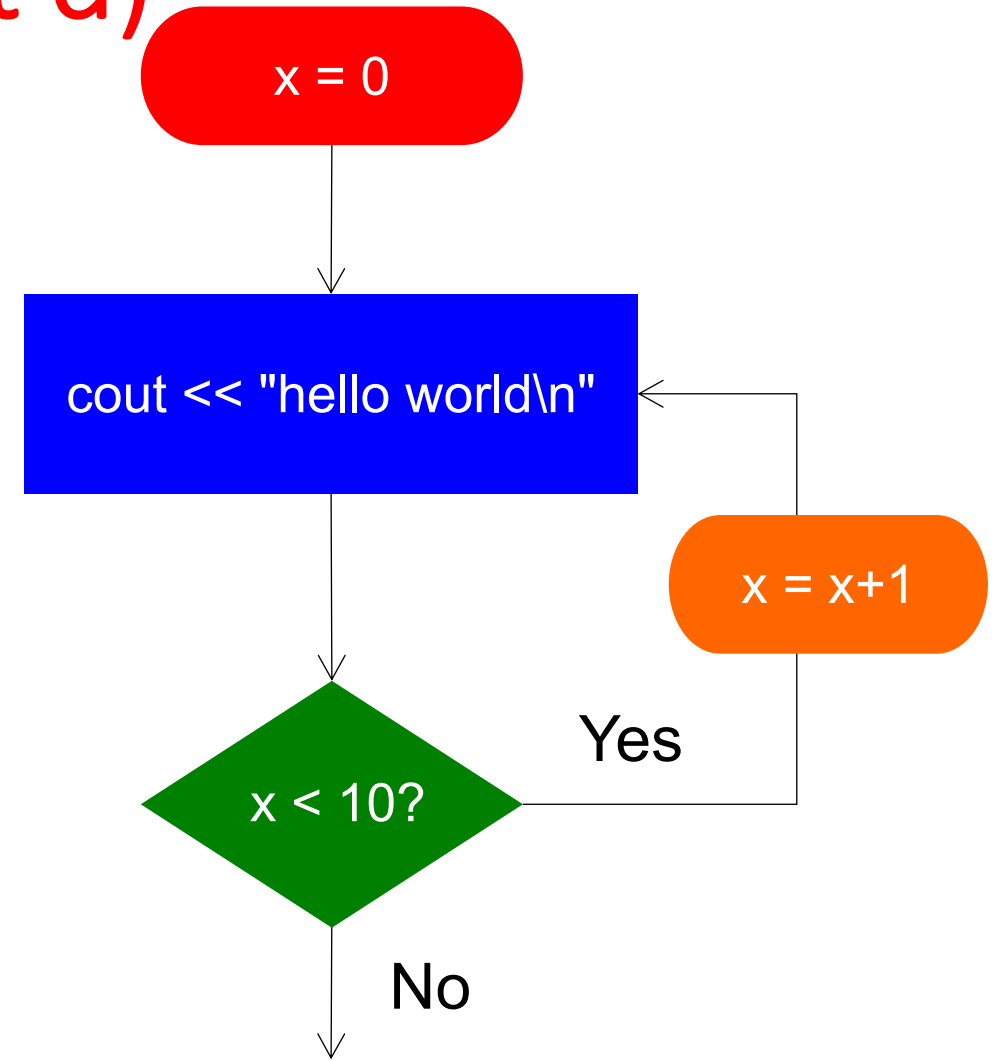
- In general, a loop consists of four parts

initialization statements

body

loop condition

post loop statements (stepping forward to exit loop condition)



Quick Review: while

- Syntax

```
while (expression)
{
    loop statement(s);
}
```

- Semantics

- If the value of expression is non-zero (true), loop statements will be executed, otherwise, the loop terminates
- After loop statements are executed, the expression will be tested again

Quick Review: do-while

- Syntax

```
do {  
    loop statement(s);  
}  
while (expression);
```

- Semantics

- loop statements are executed first; thus the loop body will be executed for **at least once**
- If the value of expression is non-zero (true), the loop repeats; otherwise, the loop terminates

Quick Review: while vs do-while

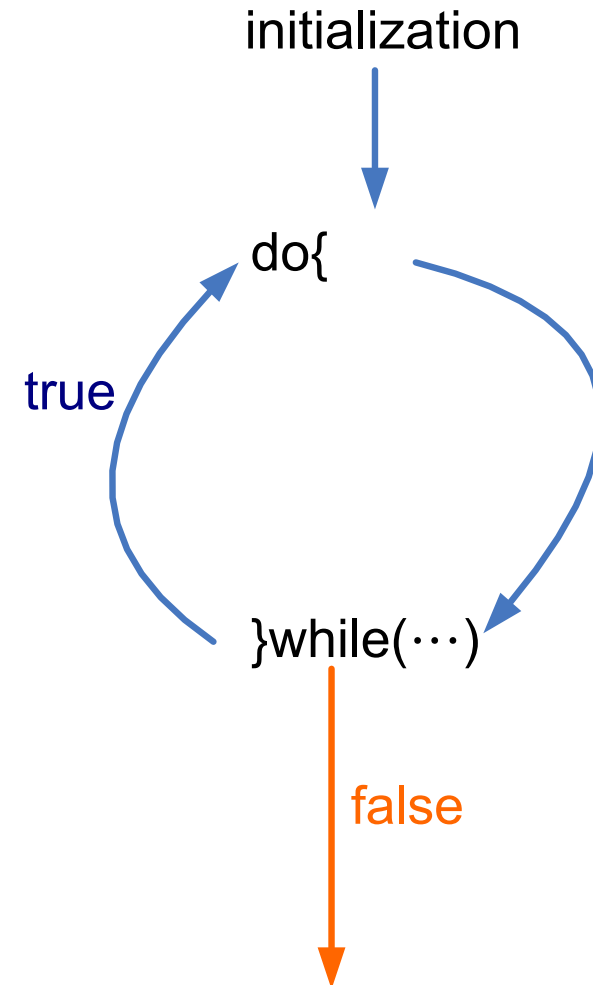
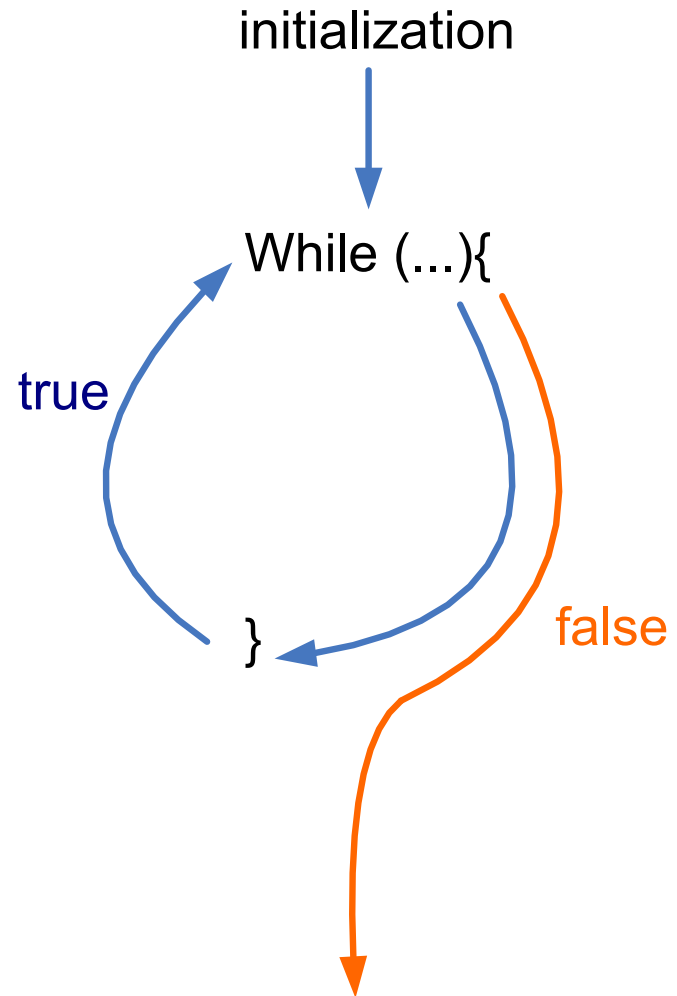
```
int x, max;
max = 0;
cout << "Enter a positive integer. ";
cout << "Type 0 to quit.\n";
cin >> x;
while (x != 0) {
    if (x > max)
        max = x;
    cout << "Enter a positive integer. ";
    cout << "Type 0 to quit.\n";
    cin >> x;
}
```

```
int x, max;
max = 0;

do {
    cout << "Enter a positive integer. ";
    cout << "Type 0 to quit.\n";
    cin >> x;
    if (x > max)
        max = x;
} while (x != 0);
```

- do-while is better suited for loops that require at least one iteration

Quick Review: while vs do-while



Quick Review: for: Syntax

```
for (expr1; expr2; expr3) {  
    loop statements;  
}
```

- Semantics

Loop statements are repeatedly executed as long as **expr2** is non-zero (true). Otherwise, the loop ends.

expr1: executed before entering the loop body. Often used for initializing a loop counter or loop status.

expr3: executed after each iteration of the loop body. Often used to update the loop counter or loop status.

Quick Review: for: Syntax (cont'd)

```
for (expr1; expr2; expr3) {  
    loop statements;  
}
```

- **expr1** and **expr3** can contain multiple statements. Each statement is separated by a comma ','
- Example

```
for (int i=0, j=0; i<10; i++, j++)  
    ...
```

Quick Review: break Statement

- The break statement causes an exit from the **innermost** enclosing loop or switch statement

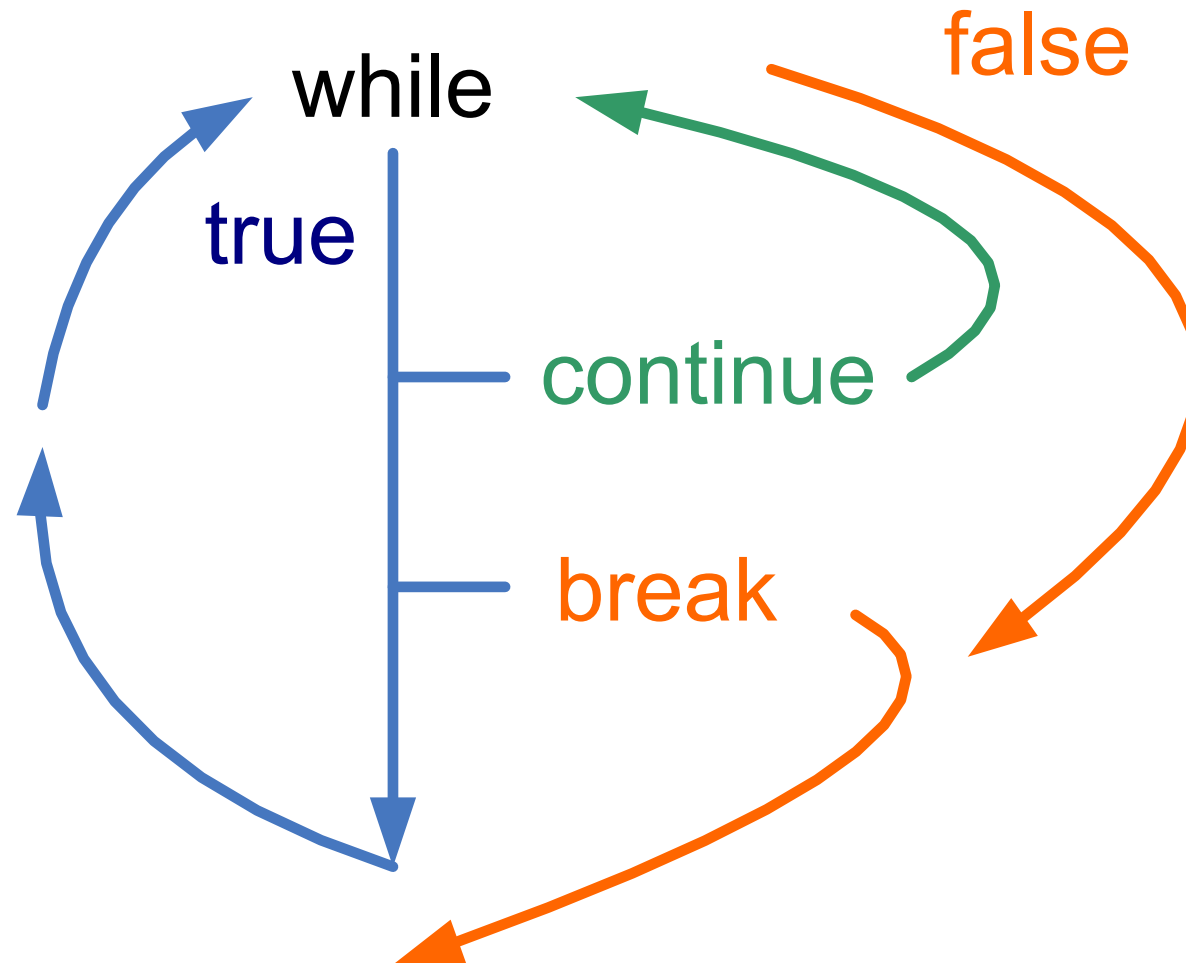
```
while (1) {  
    cin >> n;  
    if (n < 0)  
        break;  
    cout << n << endl;  
}  
// if break is run, jumps to here
```

Quick Review: continue Statement

- ❑ `continue` statement causes the `current` iteration of a loop to `stop` and the `next` iteration to begin immediately
- ❑ It can be applied in a `while`, `do-while` or `for` statement

```
cnt=0;
while (cnt<10) {
    cin >> x;
    if (x > -0.01 && x < 0.01)
        continue; // discard small values
    ++cnt;
    sum += x;
}
```

Quick Review: continue, break



Exercises: Loop 2

- write a program to produce a $n \times n$ matrix (n is input by user) with 0's down the main diagonal, 1's in the entries just above and below the main diagonal, 2's above and below that, etc.
- *hint*: consider using nested for-loop, with the outer loop responsible for row and the inner loop for each column

Example 1	Example 2
Enter the number of rows: <u>5</u> 0 1 2 3 4 1 0 1 2 3 2 1 0 1 2 3 2 1 0 1 4 3 2 1 0	Enter the number of rows: <u>8</u> 0 1 2 3 4 5 6 7 1 0 1 2 3 4 5 6 2 1 0 1 2 3 4 5 3 2 1 0 1 2 3 4 4 3 2 1 0 1 2 3 5 4 3 2 1 0 1 2 6 5 4 3 2 1 0 1 7 6 5 4 3 2 1 0
Example 3	Example 4
Enter the number of rows: <u>0</u> Please enter positive integer.	Enter the number of rows: <u>3</u> 0 1 2 1 0 1 2 1 0

```
int n;  
cout << "Enter the number of rows: ";  
cin >> n;  
if (n <= 0) {  
    cout << "Please enter positive integer.\n";  
} else {  
    // Your codes ...  
}
```

Exercises: Loop 2

```
int n;
cout << "Enter the number of rows: ";
cin >> n;
if (n <= 0) {
    cout << "Please enter positive integer.\n";
} else {
    for (int row=0; row<n; row++) {
        for (int col=0; col<n; col++)
            cout << abs(col-row) << " ";
        cout << endl;
    }
}
```

Review Exercises 1

```
int n = 1024;
int log = 0;
int i;
for (cin >> i; i < n; i = i * 2);
    log++;
cout << n << " " << log << endl;
```

What are the outputs if the input is 0?

What are the outputs if the input is 1?

Review Exercises 2

```
int number, sum = 0, count = 0;
while (++count <= 4) {
    cin >> number;
    if (number >= 0) {
        cout << "Input Ends at " << count << endl;
        break;
    }
    sum = sum + number;
}
cout << sum << " is the sum of the first " << (count - 1) << " numbers \n";
```

Assume the input is -1 -2 -3 4
What are the outputs?

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**

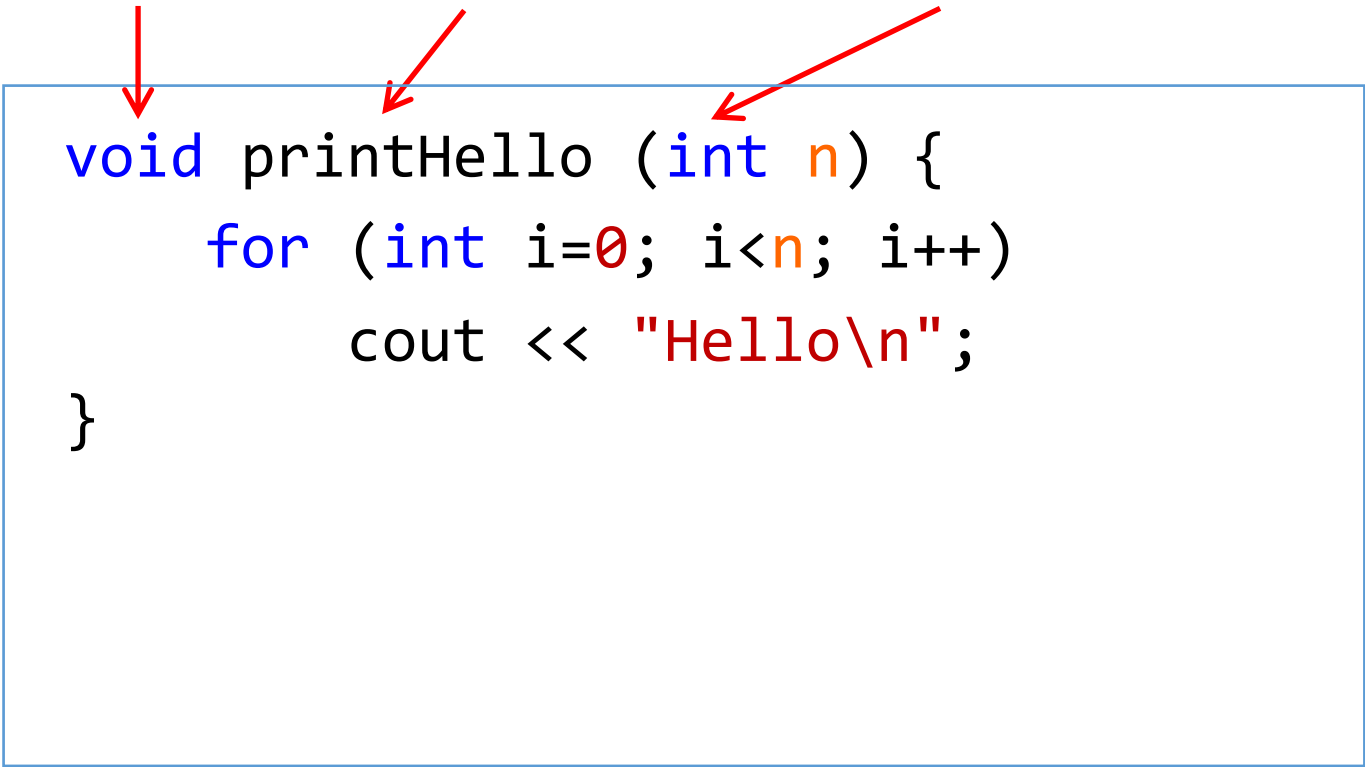


Today's Outline

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

Defining a Function

return_type function_name parameter_list

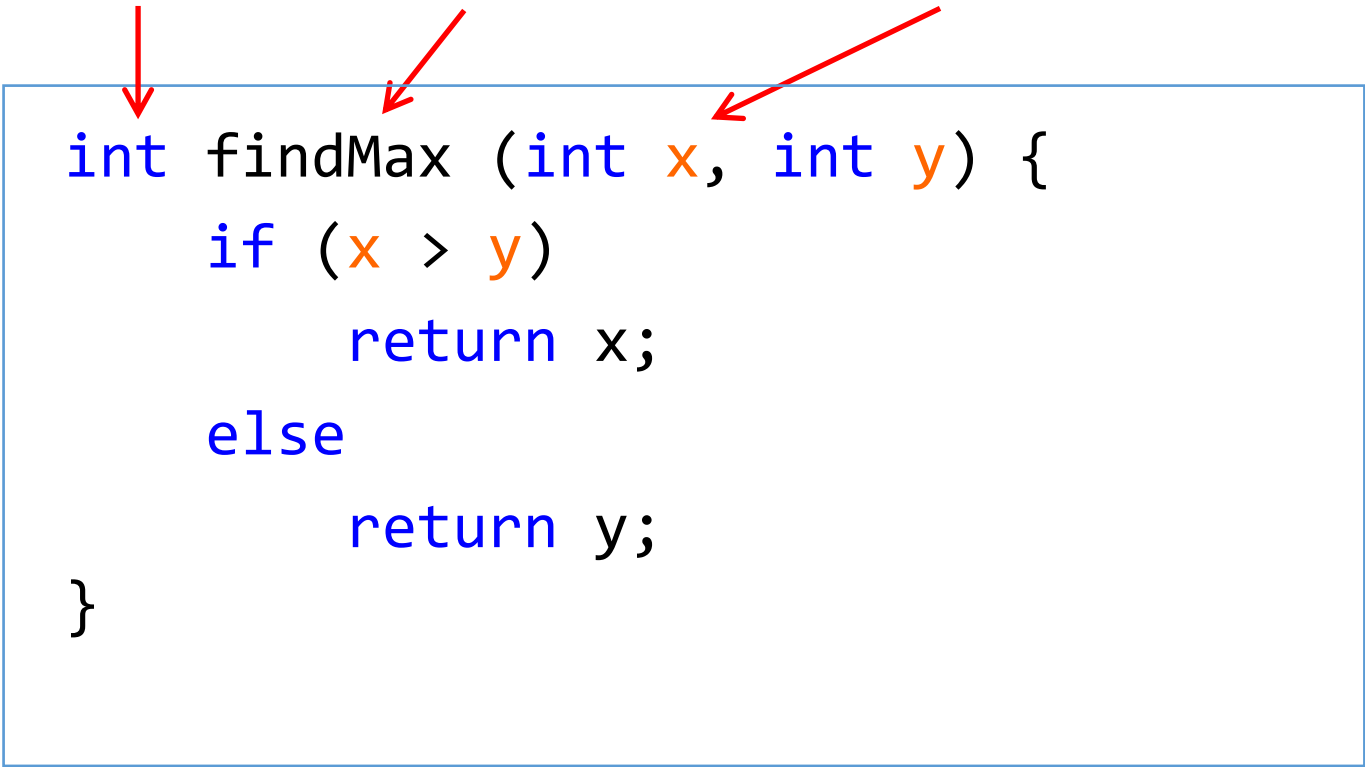


```
void printHello (int n) {  
    for (int i=0; i<n; i++)  
        cout << "Hello\n";  
}
```

- **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 function_name parameter_list

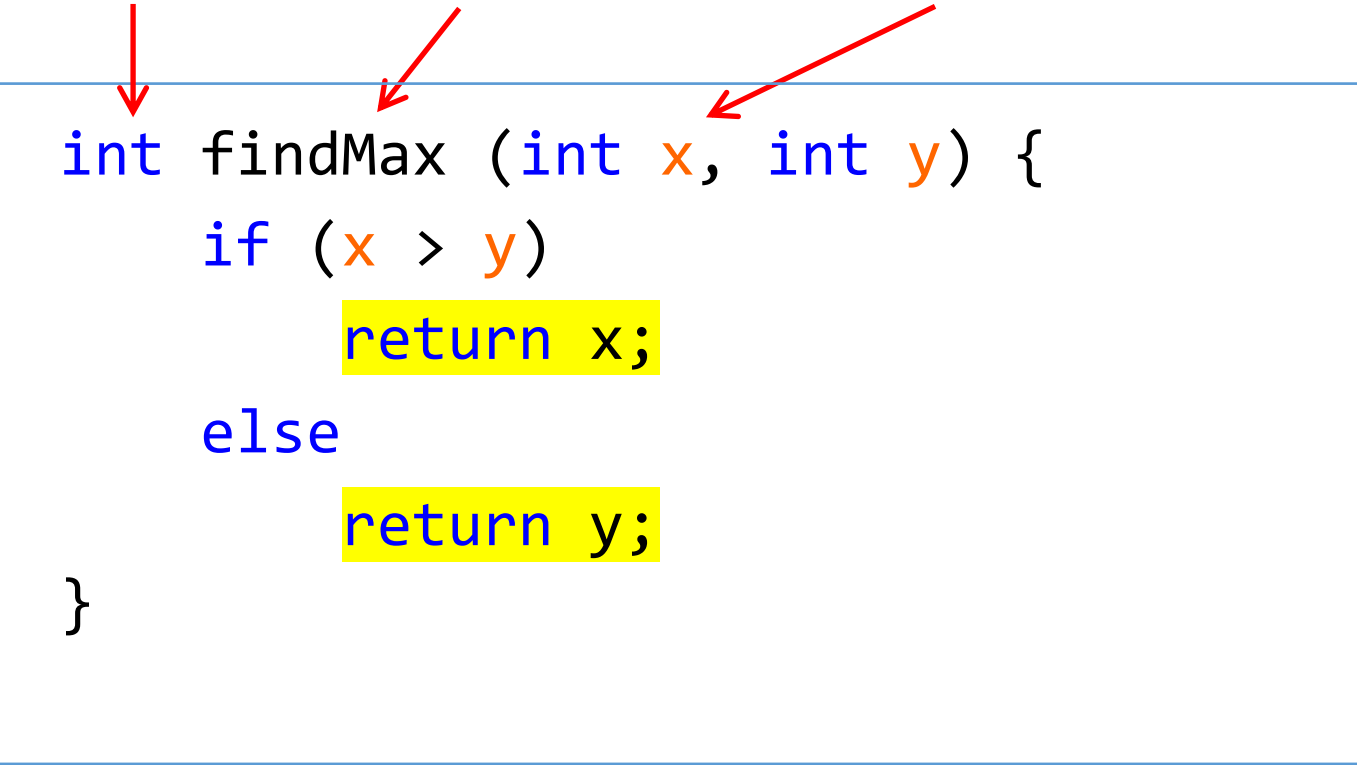


```
int findMax (int x, int y) {  
    if (x > y)  
        return x;  
    else  
        return y;  
}
```

- **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 function_name parameter_list



```
int findMax (int x, int y) {  
    if (x > y)  
        return x;  
    else  
        return y;  
}
```

- **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

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

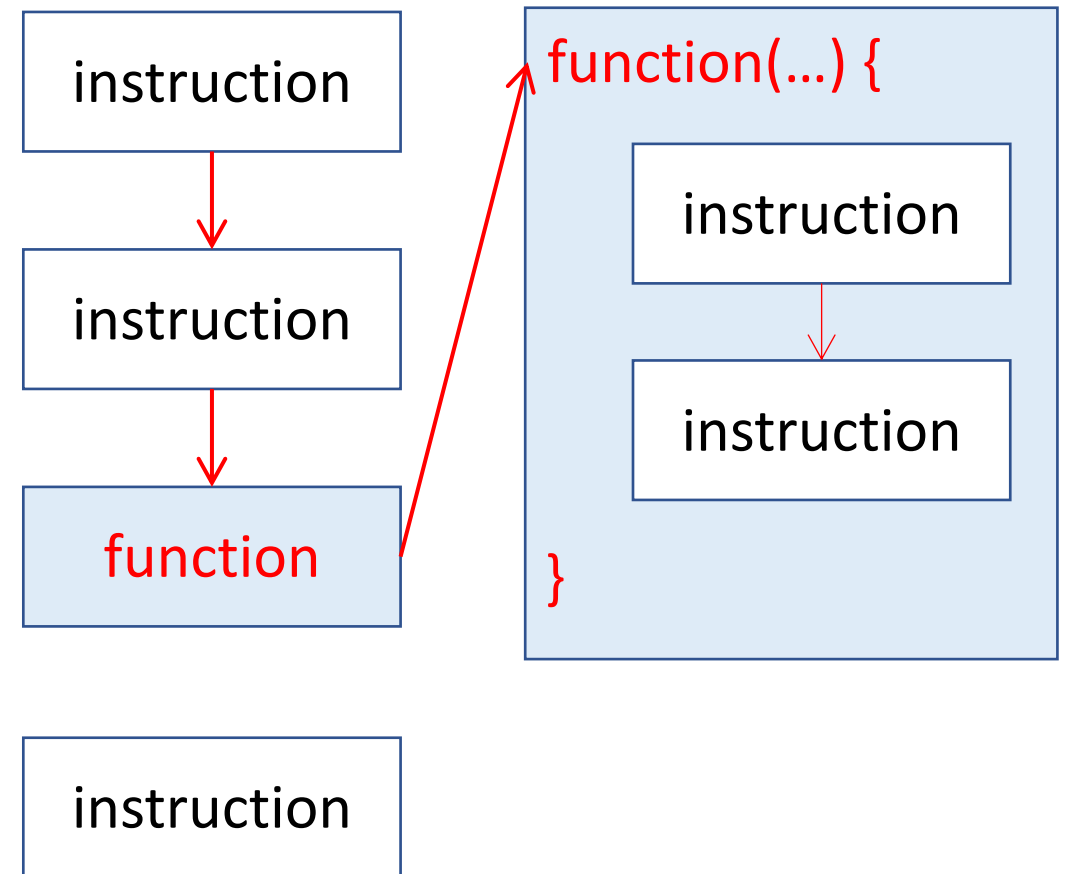
    cout << "The square has perimeter: " << 4*side;
    return 0;
}
```

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

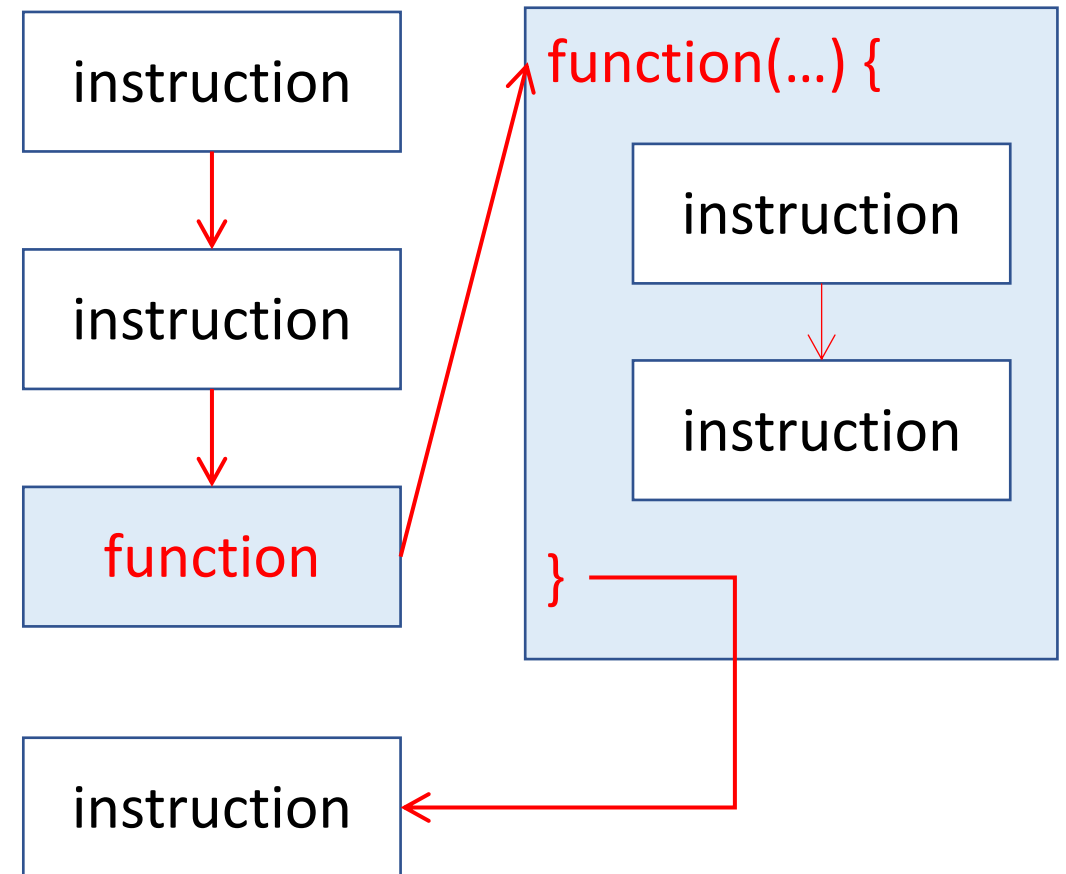
Flow of Function Call

- 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

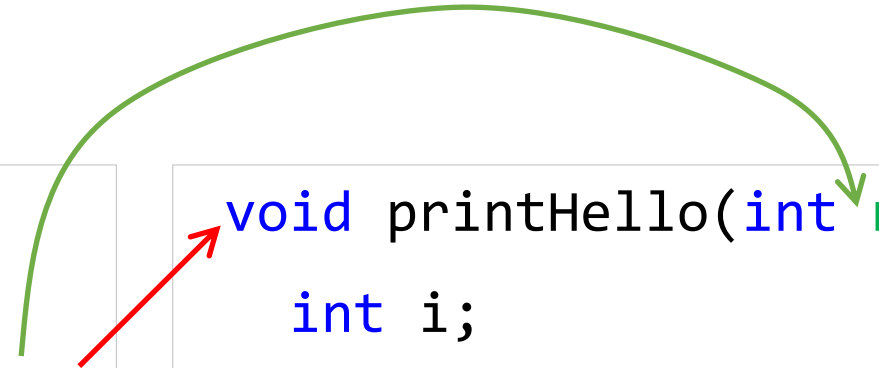


Flow of Function Call

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

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

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



Function in Memory Stack

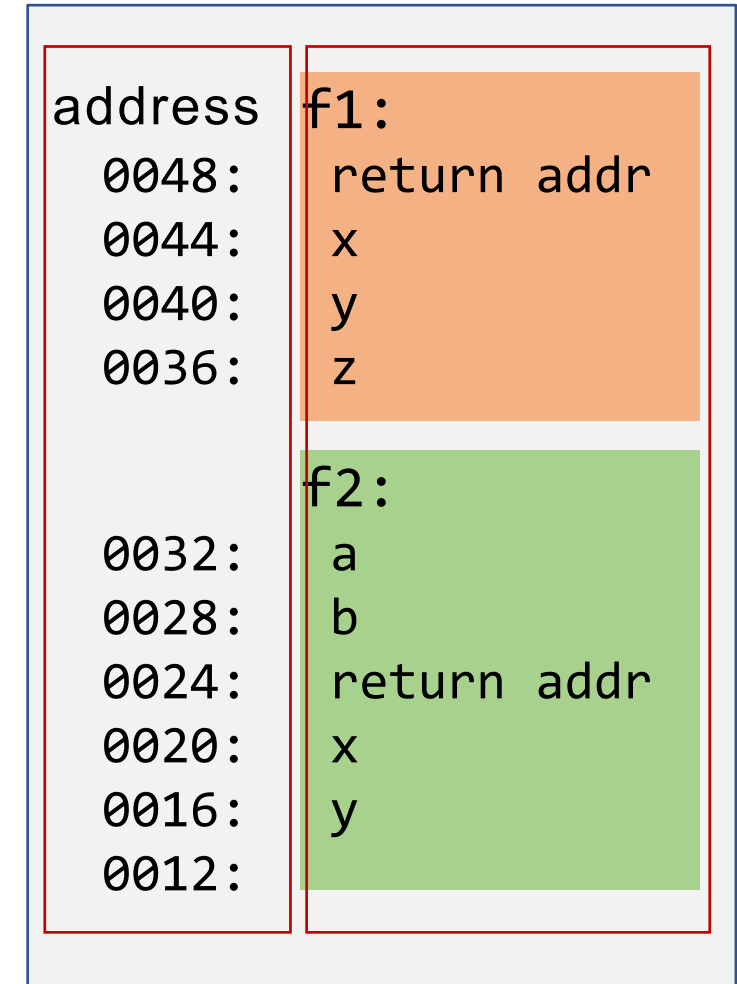
```
void f1() {  
    int x = 5;  
    int y = 6;  
    int z = f2(x, y);  
    cout << a << " " << z;  
}
```

```
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)
- 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;  
}
```

- 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;
}
```

```
// 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 ?
- 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)
- 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 (later)

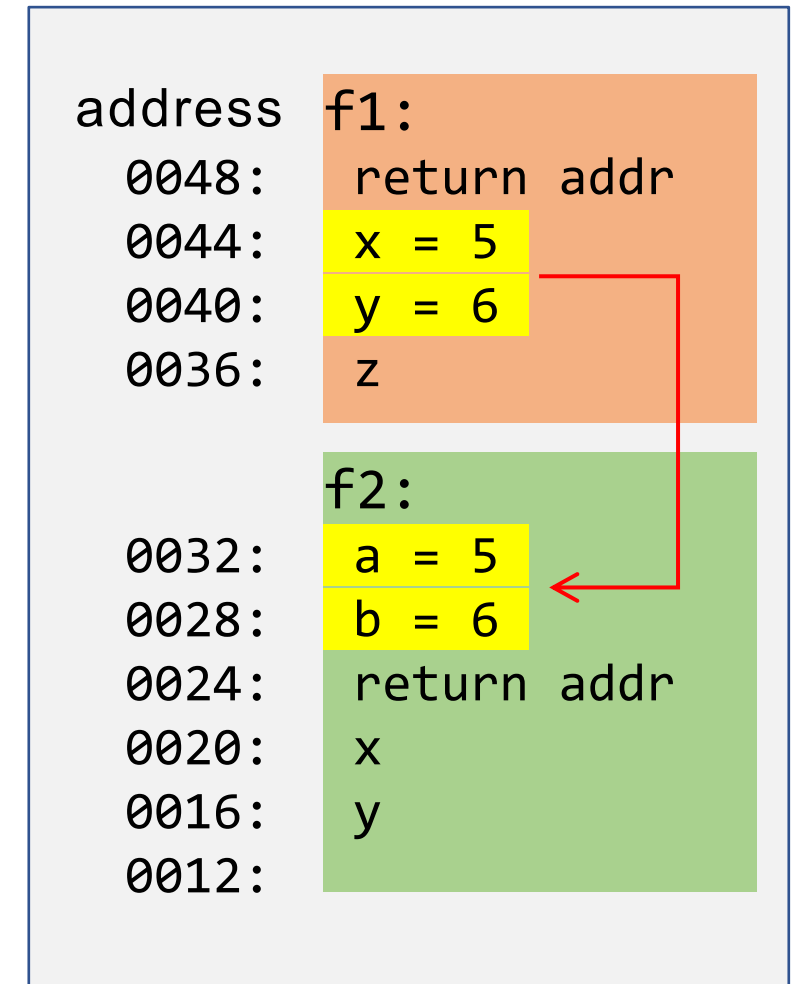
Pass-by-Value

- 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;  
}
```

Memory Stack



Pass-by-Value

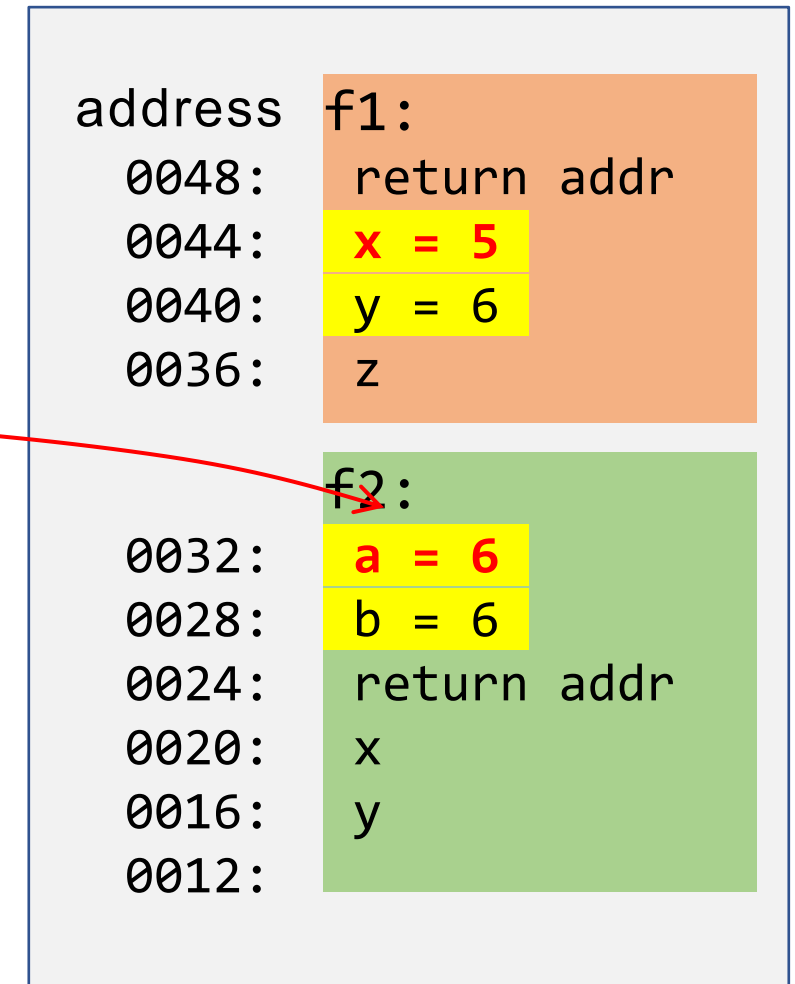
- the value of argument is *copied* to parameter

```
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

Memory Stack



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 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  
    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 << x << " " << 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

address	f1:
0048:	return addr
0044:	x = 6
0040:	y = 6
0036:	z
	f2:
0032:	a:0044
0028:	b:0040
0024:	return addr
0020:	x
0016:	y
0012:	

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); // will print 1 and 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

$$0! = 1$$

$$n! = n * (n-1) * \dots * 2 * 1, \text{ for } n > 0$$

- A recurrence relation: (induction)

$$n! = n * (n-1)!, \quad \text{for } n > 0$$

- 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;  
}
```

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
- Example:

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

Example: Vertical Number

- How to break down a number into separated digits?

```
void printDigit(int n) {  
    do {  
        cout << n%10 << endl;  
        n/=10;  
    } while (n>0);  
}
```

Input	Output
7894	4 9 8 7

Example: Vertical Number

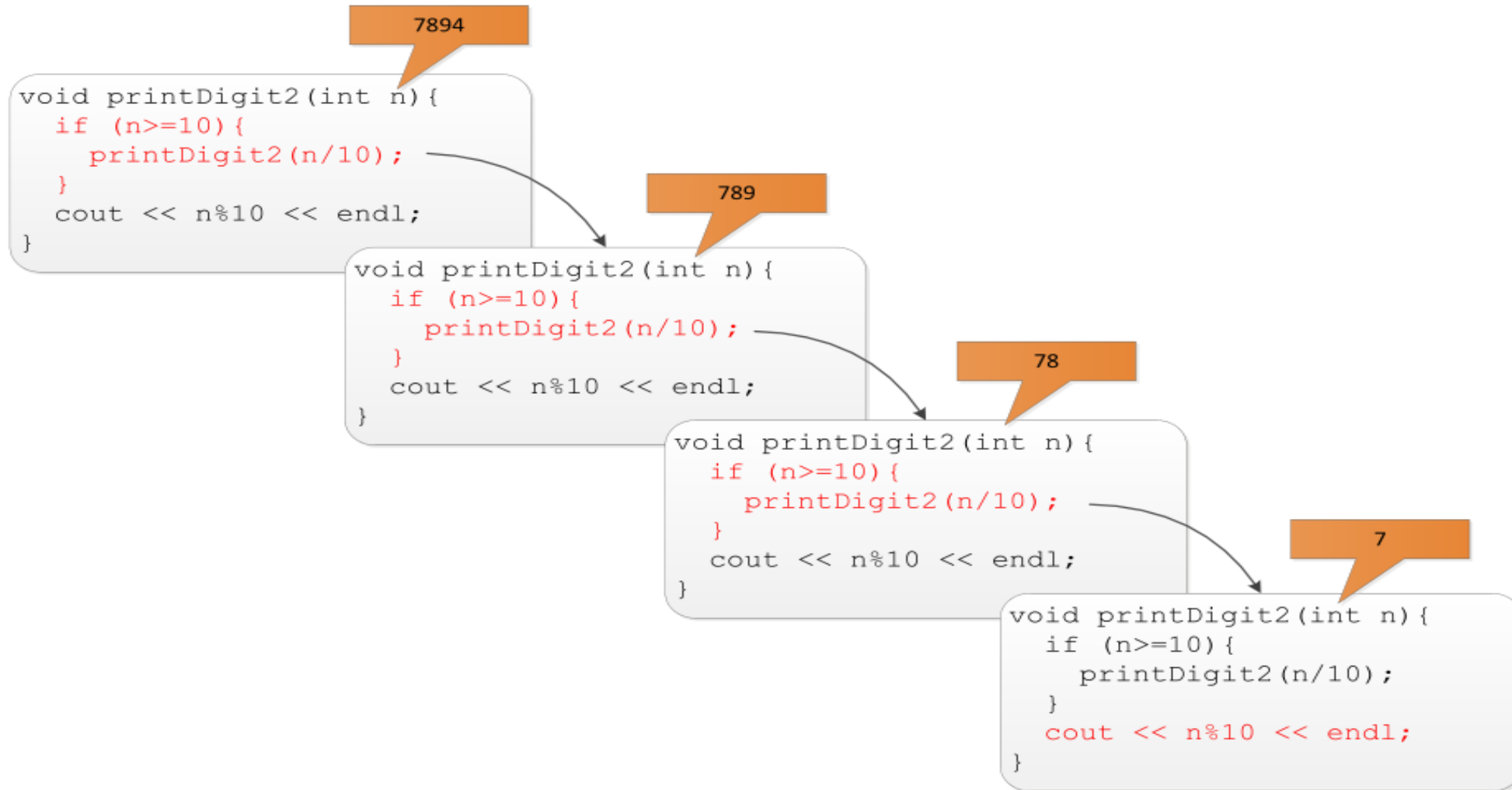
- How to break down a number into separated digits?

```
void printDigit(int n) {  
    do {  
        cout << n%10 << endl;  
        n/=10;  
    } while (n>0);  
}
```

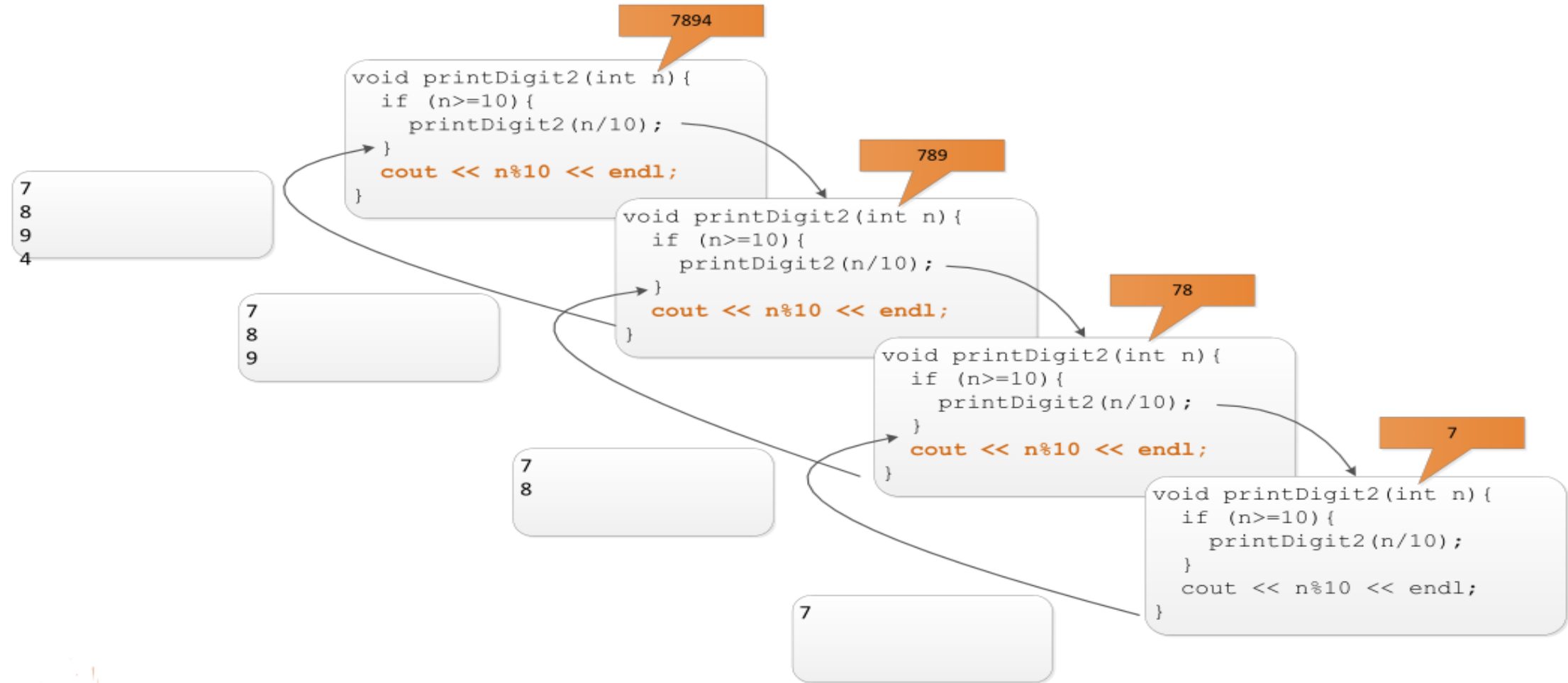
```
void printDigit2(int n) {  
    if (n>=10){  
        printDigit2(n/10);  
    }  
    cout << n%10 << endl;  
}
```

Input	Output
7894	4 9 8 7

Recursive: Entering



Recursive: Leaving



Guidelines for Recursive Functions

- Identify the **parameters**
 - e.g., n in the factorial problem
- Find out a recurrence relation between the current problem and **smaller versions** (in terms of smaller parameters) of the current problem
 - e.g., $\text{factorial}(n) = n * \text{factorial}(n-1)$
- Find out the **base cases** and their solutions
 - e.g., $\text{factorial}(0) = 1$
 - 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	$\text{factorial}(n) = n * \text{factorial}(n-1)$ e.g. $n=2 \rightarrow 2! = 2 * 1!$ $n=3 \rightarrow 3! = 3 * 2!$ $n=4 \rightarrow 4! = 4 * 3!$	$\text{printDigits}(n/10);$ $\text{cout} \ll n \% 10;$ e.g. $n=78 \rightarrow 7$ was printed $n=789 \rightarrow 7, 8$ were printed $n=7894 \rightarrow 7, 8, 9$ 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

Summary

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