Basis and Practice in Programming

SUNG KYUN KWAN UNIVERSITY(SKKU)

Programming Basics 3

Instructor Younghoon Kim

Demo for Recap

- **Make a program that**
  - receives a number from a user
  - and prints out all the divisors of the input

```
(intentionally blank box)
```

Show divisors

DEMO

```c
#include <stdio.h>

int main()
{
  int number = 0;
  printf("Input a number: ");
  scanf("%d", &number);

  for(int i = 1; i <= number; i++) {
    if(number % i == 0) {
      printf("%d ", i);
    }
  }

  printf("\n");
}
```

Basis and Practice in Programming

SUNG KYUN KWAN UNIVERSITY(SKKU)

Programming Basics 3

Instructor Younghoon Kim

Logical Operators

Boolean and Logical Operators

- **In C, Boolean is a data type that can hold true or false.**
  - Used for only those two values.
  - `_Bool` or `bool` in `stdbool.h` are supported but not commonly used.
    - since C99
  - Zero is used for false and any non-zero values are accepted as true.

- **Logical operators are for dealing true or false conditions.**
  - They return the results of Boolean operations.
  - If an operand is not Boolean, it is converted to Boolean.

Logical AND (&&) Operator

- **A binary operator that checks whether both operands are TRUE**

- **Example:**

```
if (gender == 1 && age >= 65)
    ++seniorFemales;
```

- The two conditions are evaluated first not the && operator.
  - » Precedences of == and >= are both higher than the precedence of &&.
- seniorFemale increases by one only when gender is 1 and age is greater than or equal to 65.
  - » Both conditions must be true for execution of the body.

Logical OR (||) Operator

SUNG KYUN KWAN UNIVERSITY(SKKU)

- A binary operator that checks whether either or both operands are TRUE
- Example:

```
if (semesterAverage >= 90 || finalExam >= 90)
    printf("Student grade is A");
```

- The two conditions are evaluated first not the || operator.
  - » Precedences of == and >= are both higher than the precedence of ||.
- The message will be printed out with at least one true condition.

Truth Table for && and ||

SUNG KYUN KWAN UNIVERSITY(SKKU)

- **Boolean expressions in C**
  - Expressions with relational operators, equality operators, and/or logical operators.
  - They are evaluated to one (true) or zero (false) in C.

- **Truth table for && and ||**

Expr1	Expr2	Expr1 && Expr2	Expr1 || Expr2
0	0	0	0
0	nonzero	0	1
nonzero	0	0	1
nonzero	nonzero	1	1

Short-circuit Evaluation

- **Let's reconsider the following example.**

```
if (gender == 1 && age >= 65)
    ++seniorFemales;
```

- **If gender is not equal to one, the result is determined.**
  - The combined condition is false regardless of age. C does the same thing.

- **Short-circuit evaluation**
  - The program control stops evaluating conditions with && or || when the result of the combined condition is determined.

&&, || operators

DEMO

- **Side-effects of short-circuit evaluation will be shown**

```c
#include <stdio.h>

int main()
{
  int a = 0, b = 0;

  if(++a == 0 && ++b == 1)
    printf("first if\n");
  if(++a == 2 || ++b == 2)
    printf("second if\n");
  printf("a: %d, b: %d\n", a, b);

  return 0;
}
```

Logical Negation (!) Operator

- **A unary operator used for reversing the condition**
  - True → False, False → True

```
if (!(grade == sentinelValue))
//if (grade != sentinelValue)                    // The same meaning
    printf("The next grade is %f\n", grade);
```

- **In most cases, we can write without the logical negation operator.**
- **The parentheses are necessary here.**
  - The logical negation operator has a higher precedence than == operator.
  - CHECK: What happens without the parentheses?

Logical AND (&&) Operator

- Suppose we wish to ensure that two conditions are both true before we choose a certain path of execution. In this case, we can use the logical operator && as follows:

```
if (gender == 1 && age >= 65)
    ++seniorFemales;
```

- This if statement contains two simple conditions. The condition gender == 1 might be evaluated, for example, to determine if a person is a female. The condition age >= 65 is evaluated to determine whether a person is a senior citizen. The two simple conditions are evaluated first because the precedences of == and >= are both higher than the precedence of &&. The if statement then considers the combined condition 'gender == 1 && age >= 65' which is true if and only if both of the simple conditions are true. Finally, if this combined condition is true, then the count of seniorFemales is incremented by 1. If either or both of the simple conditions are false, then the program skips the incrementing and proceeds to the statement following the if.

Logical OR (||) Operator

- Now let’s consider the || (logical OR) operator. Suppose we wish to ensure at some point in a program that either or both of two conditions are true before we choose a certain path of execution. In this case, we use the || operator as in the following program segment.

```
if (semesterAverage >= 90 || finalExam >= 90)
    printf("Student grade is A");
```

- The message “Student grade is A” is not printed only when both of the simple conditions are false (zero). The if statement considers the combined condition 'semesterAverage >= 90 || finalExam >= 90' and awards the student an “A” if either or both of the simple conditions are true.

Short-circuit evaluation

- The && operator has a higher precedence than ||. Both operators associate from left to right. An expression containing && or || operators is evaluated only until truth or falsehood is known. Thus, evaluation of the condition 'gender == 1 && age >= 65' will stop if gender is not equal to 1 (i.e., the entire expression is false), and continue if gender is equal to 1 (i.e., the entire expression could still be true if age >= 65).

- This performance feature for the evaluation of logical AND and logical OR expressions is called short-circuit evaluation.

Logical Negation (!) Operator

- C provides ! (logical negation) to enable you to “reverse” the meaning of a condition. The logical negation operator has only a single condition as an operand (and is therefore a unary operator).

- Placed before a condition when we’re interested in choosing a path of execution if the original condition (without the logical negation operator) is false, such as in the following program segment:

```
if (!(grade == sentinelValue))
    printf("The next grade is %f\n", grade);
```

- In most cases, you can avoid using logical negation by expressing the condition differently with an appropriate relational operator. For example, the preceding condition may also be written as 'grade != sentinelValue'.

Basis and Practice in Programming

SUNG KYUN KWAN UNIVERSITY(SKKU)

Programming Basics 3

Instructor Younghoon Kim

Function

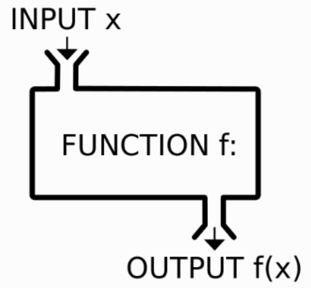
Functions in Programming - Motivation

- **Most computer programs that solve real-world problems are much larger than the programs we have seen.**

- **Divide and conquer in programming**
  - The best way to develop and maintain a large program
  - Construct a program from smaller pieces
  - Make it more manageable than the original program
  - <u>Function is a key feature for facilitating the design, implementation, operation and maintenance of large programs.</u>

Functions in Programming - Overview

- **A function**
  - is a series of statements that have been grouped together and given a name.
  - divides a program into small pieces. (easier to read, more manageable)
  - can be reused on the same (maybe similar) operations. *(reusability)*

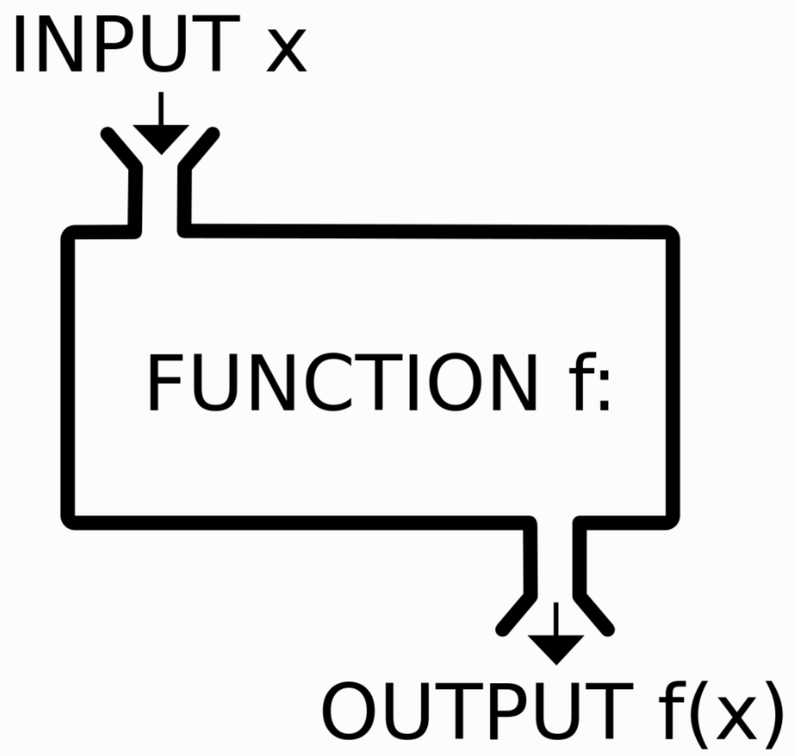


- **Inputs and outputs for functions**
  - one or more inputs and outputs
  - There can be no inputs and outputs.
    - » We will talk about `void` later in this lecture.
  - Input: Parameters, Output: Return value, Console outputs and more

What is a function in programming?

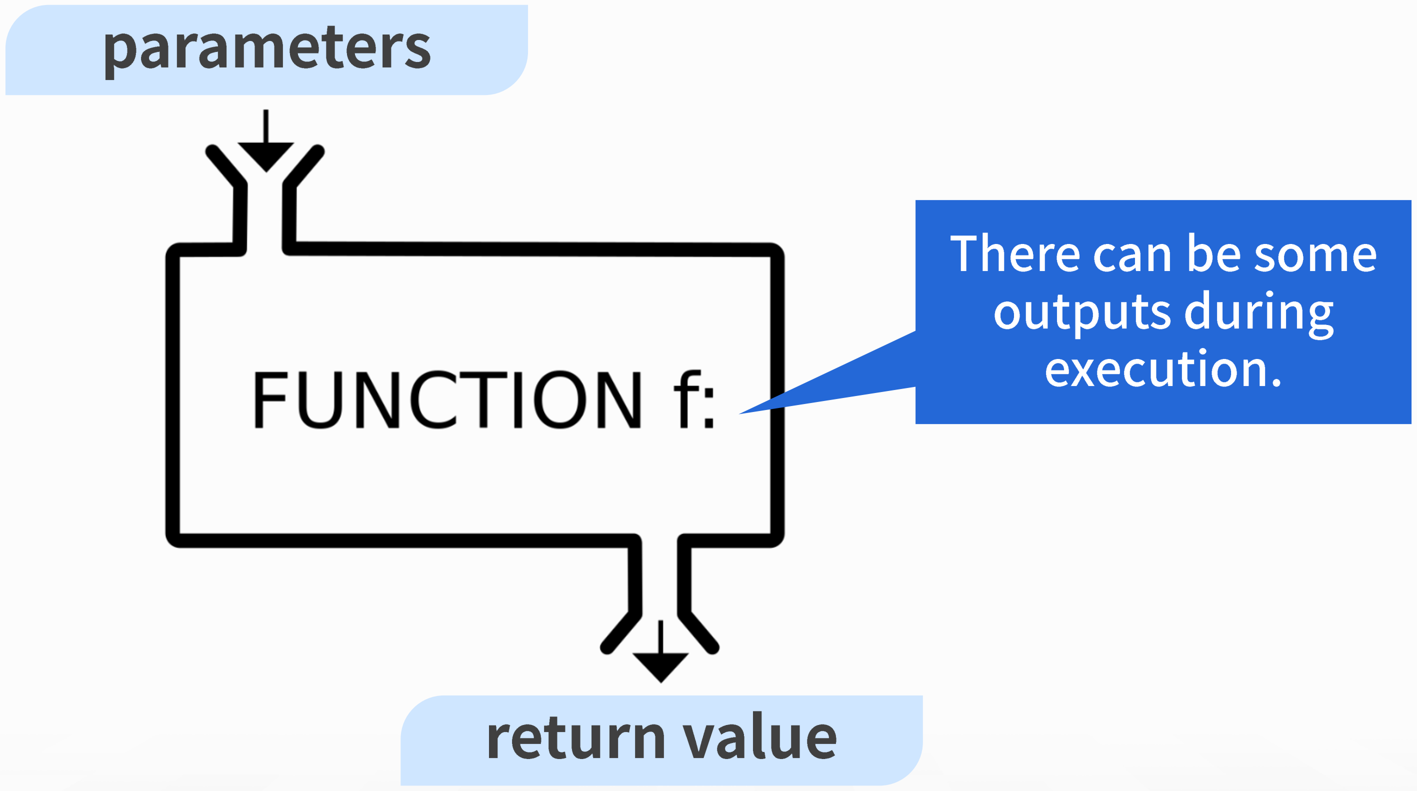
- Inputs and Outputs

$$f(x) = ax^2+bx+c$$



What is a function in programming?

- Inputs and Outputs



Function Definition in C

- Function definition = header + body

```c
int square(int number) // Header
{
        return number * number; // Body
}
```

Function Definition in C - Header

```
int square(int number) // Header
{
        return number * number; // Body
}
```

- **Return type**
  - The type of a return value
    - » Return value: a value written after the `return` keyword
  - You can specify only one return type.
- **Function name**
  - Name for calling the function (should be a any valid identifier)
  - Recommendation: Follow the popular coding style. (Ex: Google Coding Style)

Function Definition in C - Header

```
int square(int number) // Header
{
        return number * number; // Body
}
```

- **Parameter list**
  - shows information that the function expects to receive from the caller.
  - A parameter consists of a type and a parameter name.
    - » The same as a declaration of a variable without initialization.
  - There can be multiple parameters.
    - » ex) `int PrintNum(int number1, int number2)`

Function Definition in C - Body

```
int square(int number) // Header
{
        return number * number; // Body
}
```

- **Inside of a *block structure***
  - Body starts with '{' and ends with '}'.
  - There can be inner blocks.
  - Braces cannot be omitted.
- **Body consists of *statements***
  - Global rule for a program

Function Definition in C - Body

```
int square(int number) // Header
{
        return number * number; // Body
}
```

- **may return a value or not**
  - The return statement passes a value back to the calling function.
  - If there is a return value, one or more return statements can be used.
    - » With branches, there can be more than one return statements.
  - If not, a statement '`return;`' is used.
    - » It is OK to omit this return statement.

Return Type and Return Statement

- **The type of a return value must be defined in a function header.**
  - `return` *expression;* or `return;` for void return type
- **When it is set as 'void', we can omit the return statement.**
  - The void return type indicates that a function does not return a value.
  - CHECK: Compiler does not raise any errors without return statements for any functions. But the results might not be the ones we expected. So, it is STRONGLY recommended to use return statements all the time.

```
void print_hello(void)
{
    printf("hello\n");
    return; // can be omitted
}
```

Header Examples

```
void print_hello()
```

- Receives nothing and returns nothing

```
int execute_range_summation(int min, int max)
```

- Receives two integers and returns an integer

```
double average(double num1, double num2)
```

- Receives two floating point numbers and returns one

```
int main()
```

- Receives nothing and returns an integer

```
int main(void)
```

- Receives ***void***???

void keyword

- **As a function return type**
  - the **void** keyword specifies that the function does not return a value.

- **As a function's parameter list**
  - **void** specifies that the function takes no parameters.

- **In the declaration of a pointer**
  - **void** specifies that the pointer is "universal."

[Header Details] Parameters

- You can pass necessary information in the form of *parameters.*

```
double average(double a, double b)
{
        return a/b;
}
```

```
x:1.1, y:2.1
average(x, y)
average(1.1, 2.1)
average(x/2, y/2)
```

  - To call functions, use function names with proper parameters.

- Types of parameter passing
  - call by value
  - call by reference (will be covered later)
  - …

Function Examples

SUNG KYUN KWAN UNIVERSITY(SKKU)

```
int sum(int x, int y)
{
    int result = 0;
    result = x + y;
    return result;
}
```

```
int input(void)
{
    int num = 0;
    scanf("%d", num);
    return num;
}
```

```
void print(int x)
{
    int a = x;
    printf("%d", a);
    return;
}
```

```
void output(void)
{
    printf("Hello");
    printf("World");
    return;
}
```

Function Prototypes

- Similar to the header part of the function definition

```
int square(int number); // Function prototype

… (You can make use of the function square after the prototype)

int square(int number) // Header
{
        return number * number; // Body
}
```

- The compiler uses function prototypes to validate function calls.
  - return type, number of arguments, argument types, order of arguments

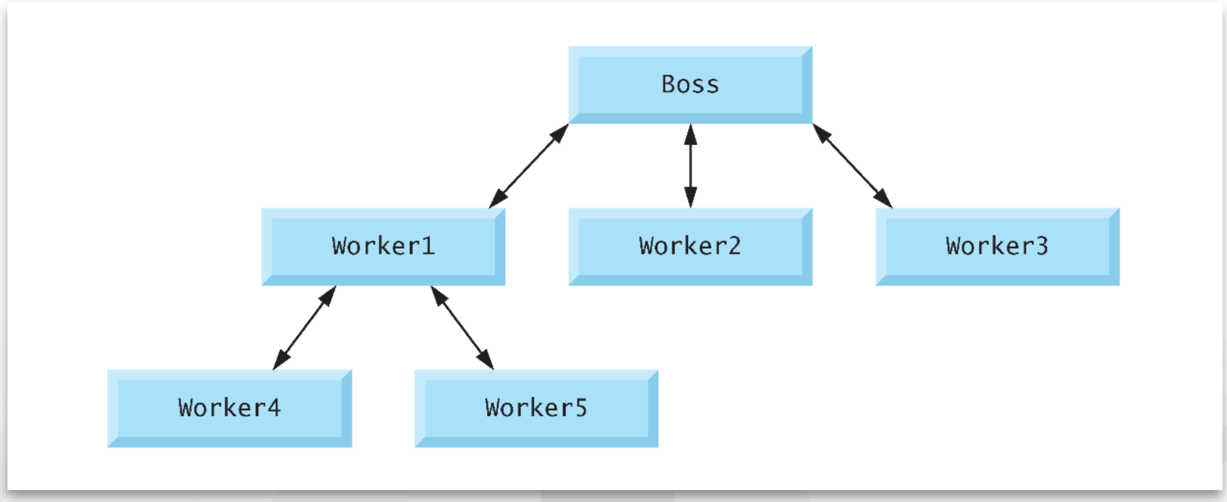
Modularizing Programs in C

- Functions are used to modularize programs. C programs are typically written by combining new functions you write with prepackaged functions available in the C standard library. The C standard library provides a rich collection of functions for performing common mathematical calculations, string manipulations, character manipulations, input/output, and many other useful operations.

- The functions printf and scanf that we've learned are standard library functions. You can write your own functions to define tasks that may be used at many points in a program. These are sometimes referred to as programmer-defined functions. The statements defining the function are written only once, and the statements are hidden from other functions. Functions are invoked by a function call, which specifies the function name and provides information (as arguments) that the called function needs to perform its designated task.

Modularizing Programs in C

- A common analogy for this is the hierarchical form of management. A boss (the calling function or caller) asks a worker (the called function) to perform a task and report back when the task is done. For example, a function needing to display information on the screen calls the worker function printf to perform that task, then printf displays the information and reports back—or returns—to the calling function when its task is completed. The boss function does not know how the worker function performs its designated tasks.



Compilation Errors

- A function call that does not match the function prototype causes a compilation error. An error is also generated if the function prototype and the function definition disagree. For example, if the function prototype had been written as

```
void maximum(int x, int y, int z);
```

and the function is written as

int maximum(int x, int y, int z) { *(lines of code)* }

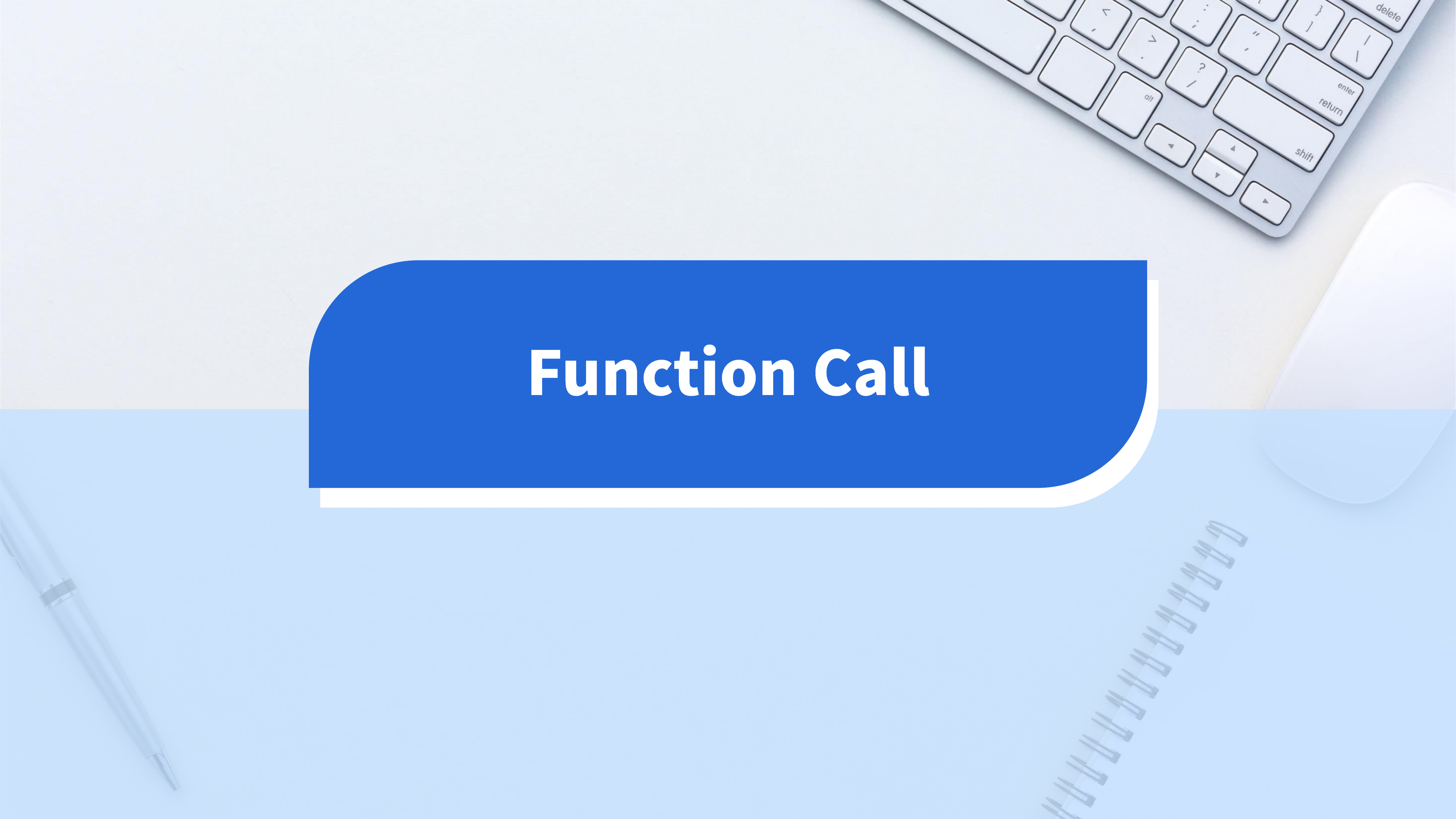
, the compiler would generate an error because the void return type in the function prototype would differ from the int return type in the function header.

Basis and Practice in Programming

SUNG KYUN KWAN UNIVERSITY(SKKU)

Programming Basics 3

Instructor Younghoon Kim



Function Call

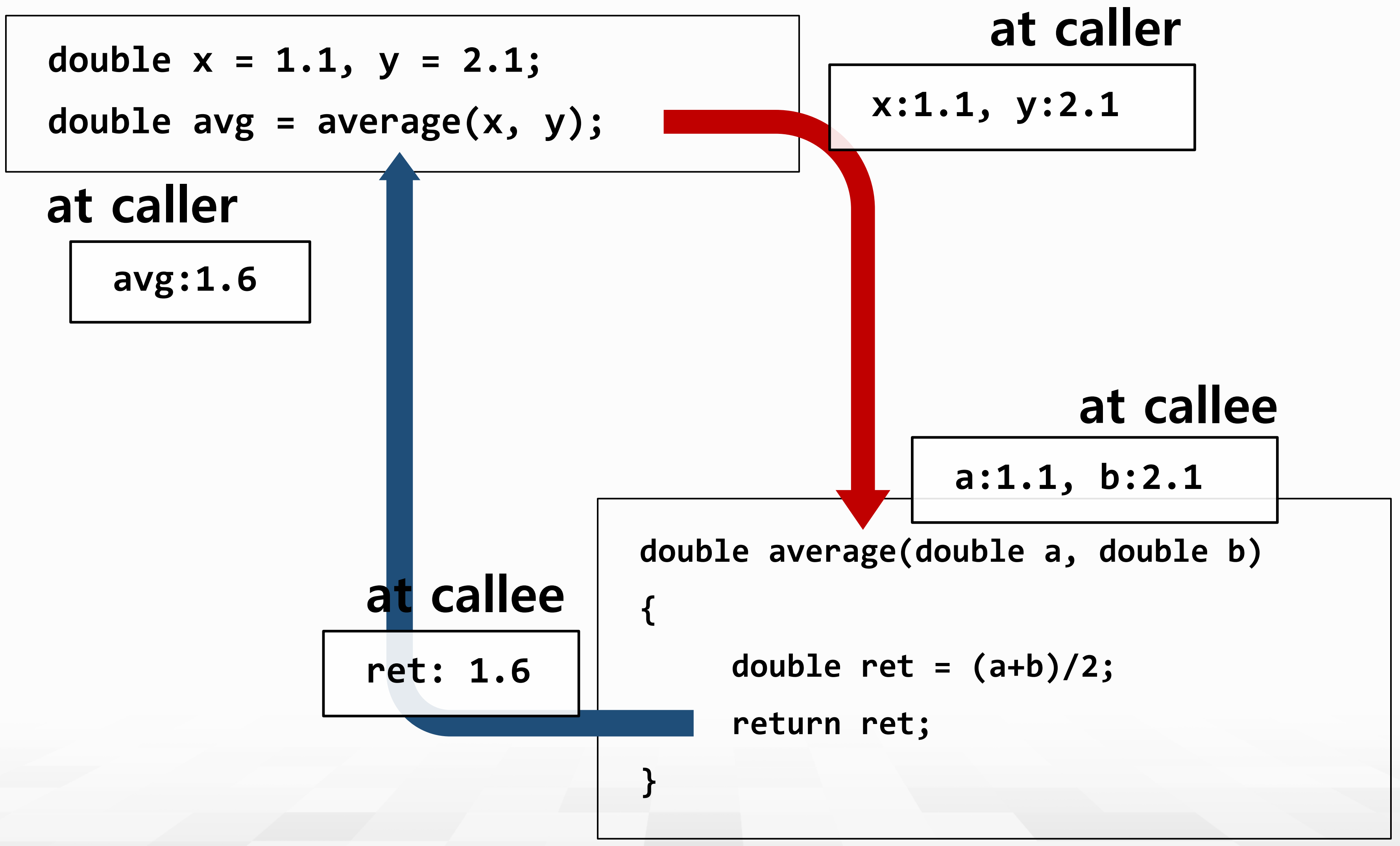
Function Calls

- To call a function, specify the function name with proper parameters.
- Control flow of a function call
  - When a function is called, the control jumps to the function definition.
  - It returns when it ends or when the program meets a return statement.
  - The return value, if any, substitutes the function call for afterward usage.
- Return value usage: You can store it to a variable or discard it after using.

```
double x = 1.1, y = 2.1;
double avg = average(x, y); // OK!

printf("Average: %f\n", average(x, y));  // OK!!
```

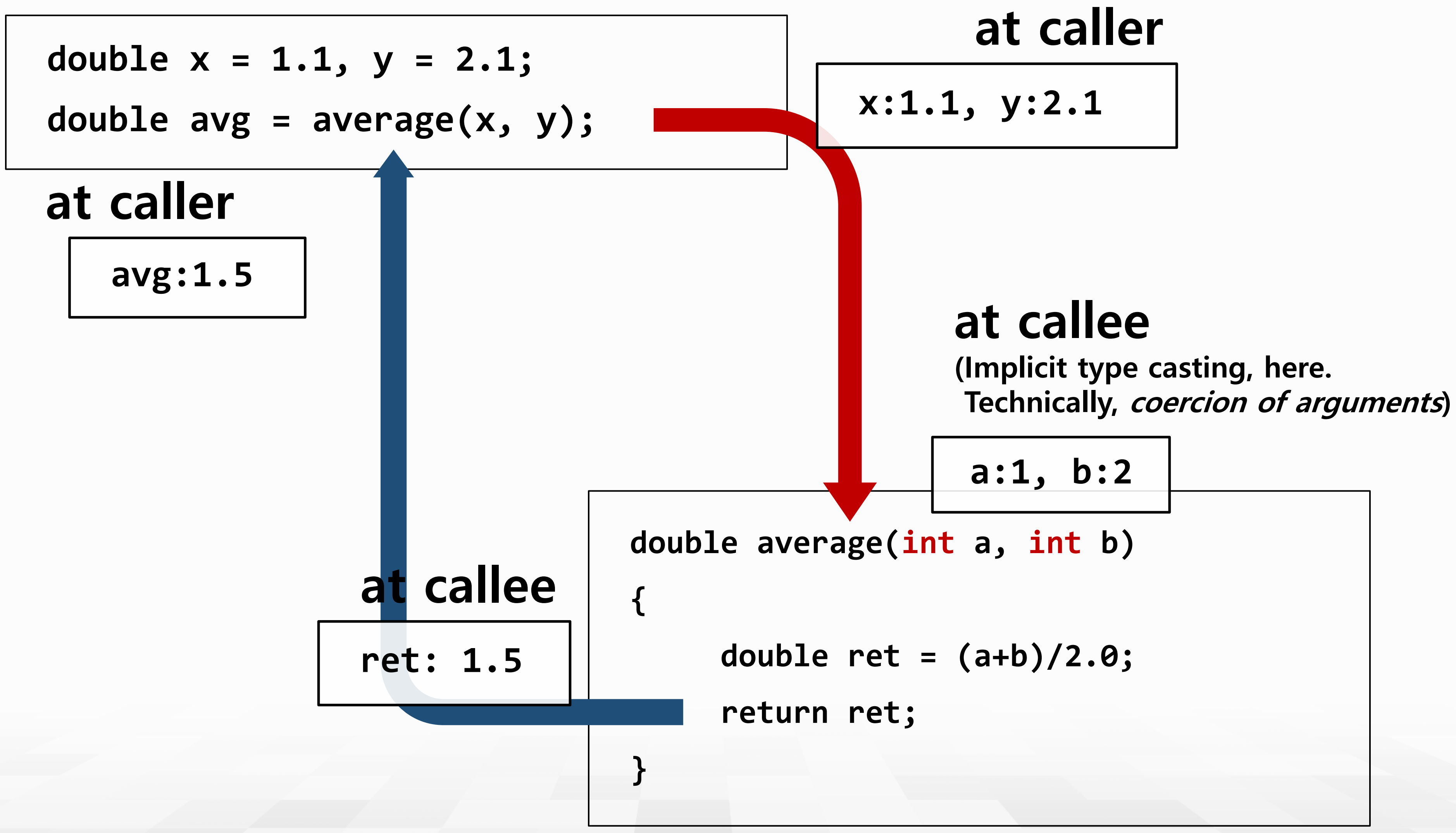
Parameter Passing Example 1



```
double x = 1.1, y = 2.1;
double avg = average(x, y);
```

```
double average(double a, double b)
{
    double ret = (a+b)/2;
    return ret;
}
```

Parameter Passing Example 2


```
double x = 1.1, y = 2.1;
double avg = average(x, y);
```

```
double average(int a, int b)
{
      double ret = (a+b)/2.0;
      return ret;
}
```

Function call example

DEMO

- Check the control flow with a function call
  - Variations with other types will be shown.

```c
#include <stdio.h>
double avg(int x, int y) {
  return (x + y)/2.0;
}

int main() {
  int num1, num2;
  printf("Input two numbers(ex: 1 2): ");
  scanf("%d %d", &num1, &num2);
  printf("Average of %d and %d is %f.\n", num1, num2, avg(num1, num2));
  return 0;
}
```

Variable Scope

- **Local Variables**
  - Most of variables you are using and will use are ***locally*** valid.
  - The term ***'locally'*** usually and roughly means ***inside of a block.***

```
int main(void)
{
        int i;
        int i; // ERROR
}
```

```
int main(void)
{
        { int i; }
        { int i; } // OK
        i=1;       // ERROR
}
```

  - **All variables defined in function definitions** are local variables
  - **A function's parameters** are also local variables of that function.

- **Global Variables (will be covered later)**

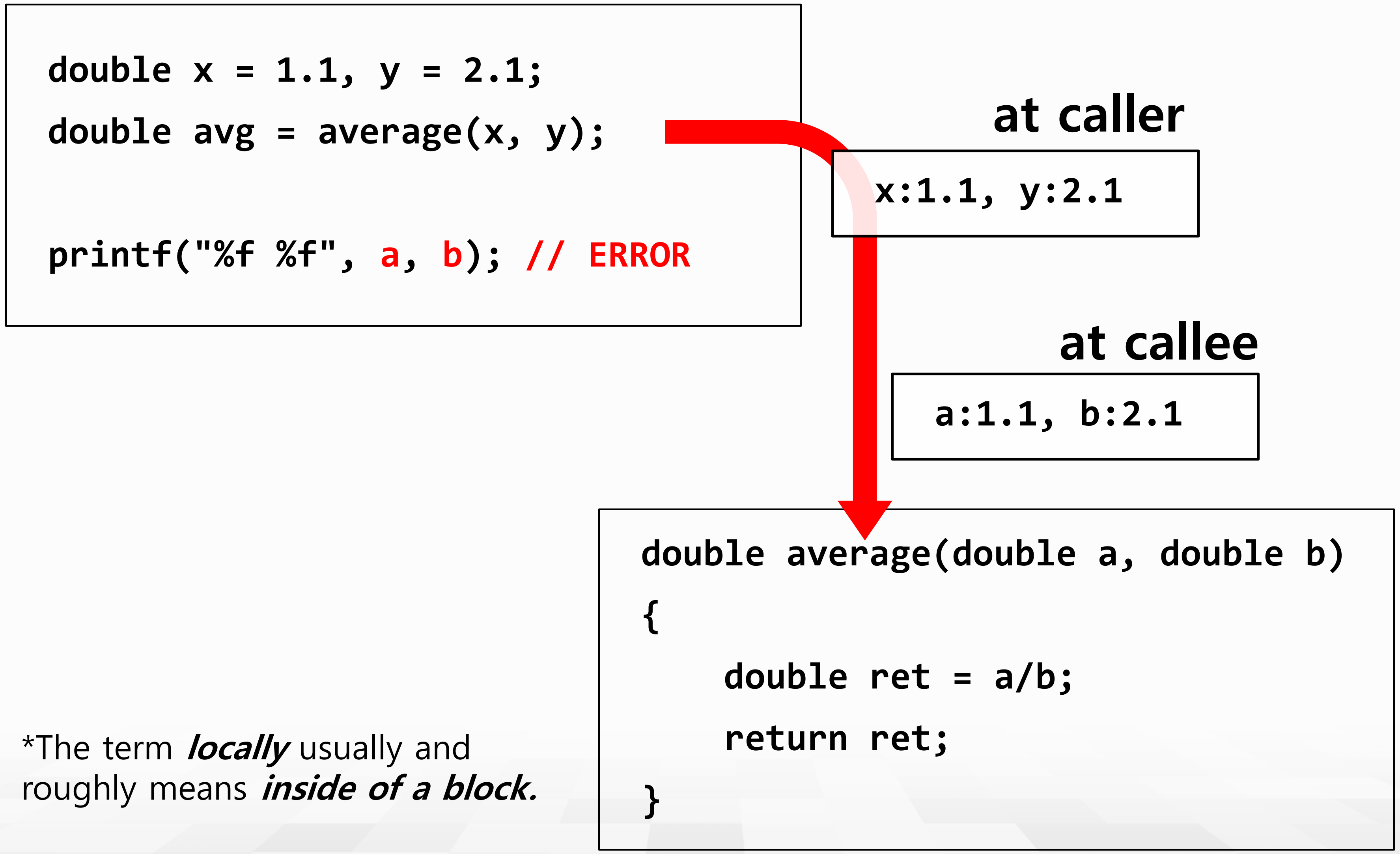
Variable Scope

- **Variable shadowing**
  - occurs when the same name is used by variables declared in an inner and the outer scope
  - The variable in the inner scope masks or shadows the variable in the outer scope.
  - Variable shadowing can occur even with different data types.

```
int main(void)
{
	int i = 1;
	{ int i = 2; } // OK
	printf("%d\n", i); // OK! Value?
}
```

```
int main(void)
{
	int i = 1;
	{ double i = 2; } // OK
	i=1;
}
```

Scope for Parameters



```
double x = 1.1, y = 2.1;
double avg = average(x, y);

printf("%f %f", a, b); // ERROR
```

```
double average(double a, double b)
{
    double ret = a/b;
    return ret;
}
```

*The term *locally* usually and roughly means *inside of a block.*

Function call example (an error case)

DEMO

```
#include <stdio.h>

double avg()
{
  return (num1 + num2)/2.0;
}

int main()
{
  int num1, num2;

  printf("Input two numbers(ex: 1 2): ");
  scanf("%d %d", &num1, &num2);
  printf("Average of %d and %d is %f.\n", num1, num2, avg( ));

  return 0;
}
```

Example program - Divisor War

DEMO

- Divisor War is a simple game that a number with more divisors wins the game. Make a program that receives two integers and shows the results of Divisor War game.

(Execution example)

```
Input a number1: 10
Input a number2: 12
1, 2, 5, 10
1, 2, 3, 4, 6, 12

12 wins!
```

(Execution example)

```
Input a number1: 9
Input a number2: 25
1, 3, 9
1, 5, 25

They tied!
```

```
    (intentionally blank box)
```

▪ Divisor War

DEMO

```
#include <stdio.h>

int GetNumberOfDivisors(int number)
{
  int count = 0;
  for(int i = 1; i <= number; i++) {
    if(number % i == 0) {
      if(count != 0)
        printf(", ");
      printf("%d", i);
      count++;
    }
  }
  printf("\n");

  return count;
}
```

▪ Divisor War (continued)

DEMO

```
int main()
{
  int number1 = 0, number2 = 0;
  int count1 = 0, count2 = 0;
  printf("Input a number1: ");
  scanf("%d", &number1);
  printf("Input a number2: ");
  scanf("%d", &number2);

  count1 = GetNumberOfDivisors(number1);
  count2 = GetNumberOfDivisors(number2);

  printf("\n");

  if(count1 > count2) printf("%d wins!\n", number1);
  else if(count1 < count2) printf("%d wins!\n", number2);
  else printf("They tied!\n");
}
```