C++ Programming

Ch. 7 Functions: C++'s Programming Modules

Spring 2014

Myung-Il Roh

Department of Naval Architecture and Ocean Engineering
Seoul National University

Ch. 7 Functions: C++'s Programming Modules

Contents

- ☑ Function Review
- ☑ Function Arguments and Passing by Value
- ☑ Functions and Arrays
- ☑ Functions and C-Style Strings
- ☑ Functions and Structures
- ☑ Recursion
- ☑ Pointers to Functions
- ☑ Summary
- ☑ Practice

Function Review (1/2)
- Overview

☑ Functions

- An independent unit of a program that performs a specific task.

☑ Steps for Using a User-Defined Function

☑ Defining, Prototyping, and Calling a Function

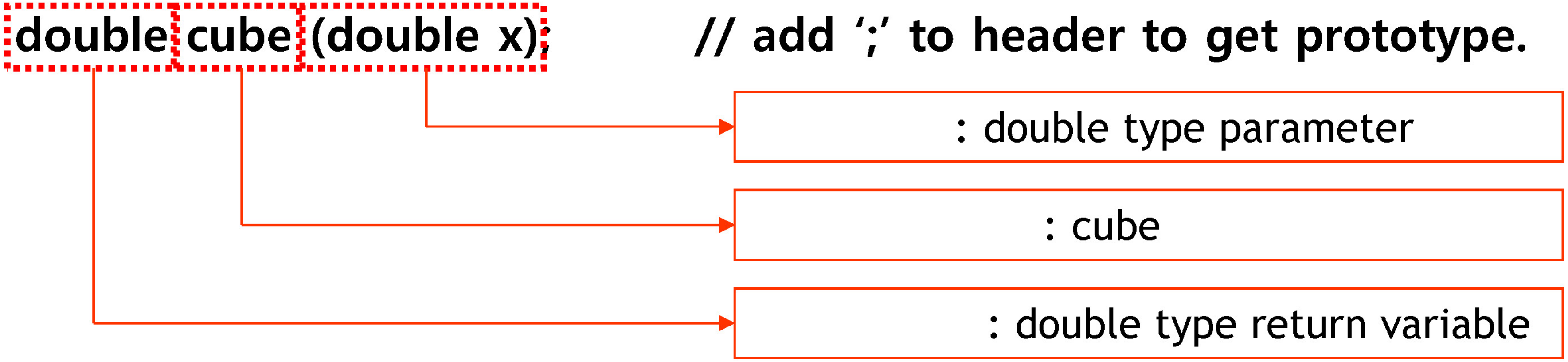
- Ex.

```
void tv();              // Provide a function prototype.

void main() {
    tv();               // Call the function.
}

void tv() {             // Provide a function definition.
...
}
```

Function Review (2/2)
- Prototyping and Calling a Function

- Describes the function interface to the compiler.
- Tells the compiler , if any, the function has.
- Tells the compiler .
- Convert the arguments to the correct type when the type of arguments is different.

☑ Ex.



```
double cube (double x);          // add ';' to header to get prototype.
```

: double type parameter

: cube

: double type return variable

```
void cheers(int);                // Okay to drop variable names
                                    in prototype.
```

Function Arguments and Passing by Value (1/4)
- Function Arguments or Parameters

☑ Classification of Arguments (or Parameters)

(‘ ’ in C++):
declared in the prototype or declaration of a function that is

(‘ ’ in C++):

☑ Argument Passing

- Assign the argument to the parameter
- That is, Actual arguments ➡ Formal parameters

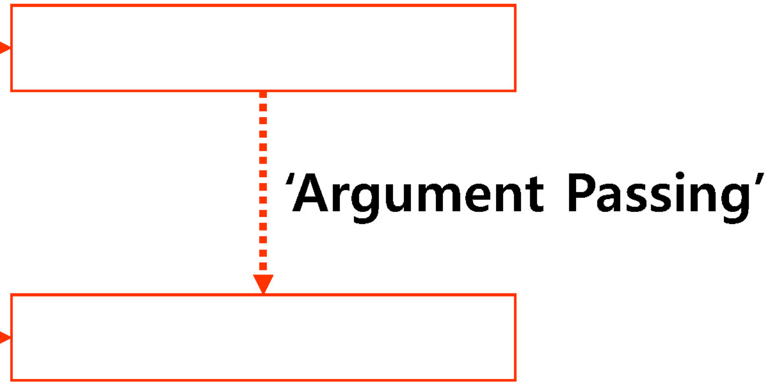
☑ Cf.

- In common usage, the argument and the parameter are often interused.

Function Arguments and Passing by Value (2/4)

☑ Ex.

```
double cube(double x);      // Provide a function prototype for a
user-defined function

int main()
{
    …
    double side = 5;
    double y = cube(side);
    …
}

double cube(double x)
{
    return x * x * x;
}
```



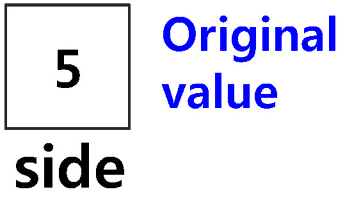
‘Argument Passing’

Function Arguments and Passing by Value (3/4)

☑ **Ex.**

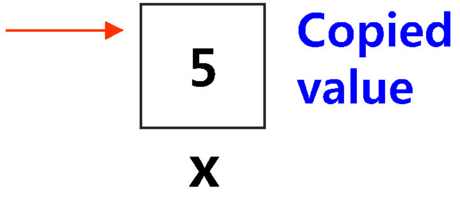
```
double cube(double x);      // Provide a function prototype for a
user-defined function

int main()
{
    …
    double side = 5;
    double y = cube(side);
    …
}

double cube(double x)
{
    return x * x * x;
}
```

Create variable called side and assign it the value 5.



Pass the value 5 to the cube() function.

Create variable called x and assign it passed value 5.



Function Arguments and Passing by Value (4/4) - Multiple Arguments

☑ Multiple Arguments

- A function can have more than one argument.

☑ Using Multiple Arguments

- Separate the arguments with commas (',').
- Cannot combine declarations of the parameters.
- Ex.

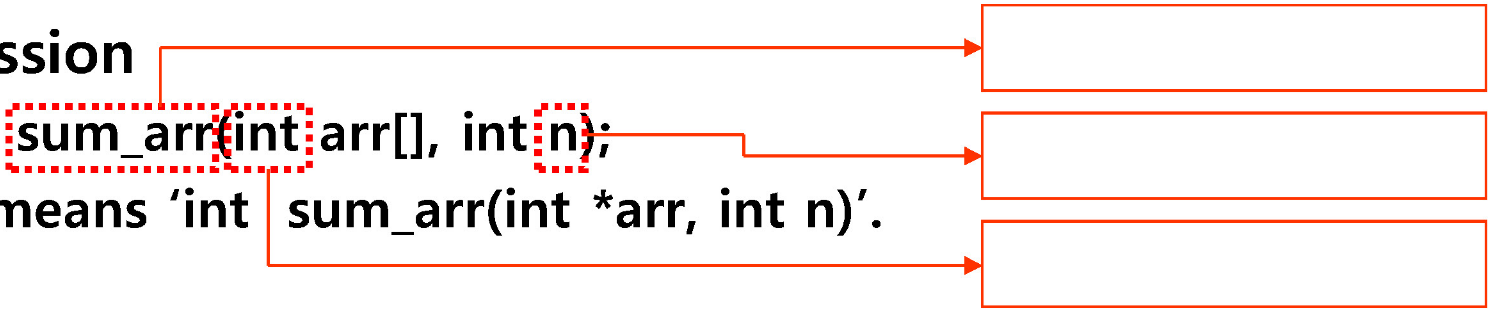
```
void n_chars(char c, int n);        //              .
void n_chars(char, int);            //              . We can drop
                                      the name of the variables.
void fifi(float a, b);              //                    .
void fifi(float a, float b);        //              .
void fifi(float, float);            //              .
```

Functions and Arrays (1/4)

☑ Using Arrays as Arguments

- Can use arrays’ name and size as formal parameter.

☑ Expression



- int sum_arr(int arr[], int n);
- It means ‘int sum_arr(int *arr, int n)’.

☑ Arrays and Pointers

- int arr[n];
  - ‘arr’ is , and of the array

Functions and Arrays (2/4)
- Additional Features of the Functions and Arrays

☑ **Implications of Using Arrays as Arguments**

- The array contents aren't really passed to the function. Instead, the function where the array is ( ), what kind of elements it has ( ), and how many elements it has ( ) are passed.
  - Unlike in case of passing an ordinary variable ( ), If we pass an array, the function accesses directly to the original array and works with it.
  - To use array addresses as arguments saves the time and memory.
  - However, it raises the possibility of inadvertent data corruption.

- Protecting array with 'const'
  - Unless the purpose of a function is to alter data passed to it, you should guard the original array from the modifying it with 'const' keyword.
  - Ex. void show_array(double arr[], int n);
    - It doesn't mean that the original arrary arr[] is constant, but it means that we can't use arr[] to change the data.

Functions and Arrays (3/4)
- Pointers and the ‘const’ Keyword

☑ The ‘const’ Pointer as Formal Argument (‘Parameter’)

- We can use a pointer as the ‘const’ argument like array by declaring formal pointer argument.

- The reason why declaring pointer argument with the ‘const’ argument
  - It prevents errors that we change the data by mistakes.

- Two ways to use ‘const’ on the pointer.
  - It prevents us from using the pointer to change the pointed-to value.
    ➡ Recommended
  - It prevents us from changing where the pointer points.

Functions and Arrays (4/4)
- Examples of Using the ‘const’ Pointer

☑ **Pointers-to const (Method 1) and const pointer (Method 2)**

```
int gorp =16;
int chips =12;
```

Pointer point to a constant object

```
          * p_snack = &gorp;

*p_snack = 20;      //
p_snack = &chips;   //
```

Pointer itself constant

```
int*                = &gorp;

*p_snack = 20;      //
p_snack = &chips;   //
```

➡ We should declare formal pointer arguments to const whenever it’s appropriate to do so.

Functions and C-Style Strings

☑ Functions with C-Style String Arguments

- An array of char
- A quoted string constant (also called a string literal)
- A pointer-to-char set to the address of a string
- Ex.

```
char ghost[15] = “galloping”;
char *str = “galloping”;
int n1 = strlen(ghost);             // ‘ghost’ is ‘&ghost[0]’.
int n2 = strlen(str);               // pointer to char
int n3 = strlen(“galloping”);       // address of string
```

All they are look like pass the array, but   of the first element of the array. ➡ char* type pointer

- Prototype of the function that uses string as argument
  - The type for the formal parameter representing a string is ‘   ’.
  - Ex.

```
int c_in_str(const char      , char ch);          // Ok
int c_in_str(const char       , char ch);         // Ok
```

[Review] Notices for Using Pointers (1/2)

☑ Like a ordinary variable that can be used after initialized, a pointer can be used after it has an specific address value.

☑ Ex. Which part of this code is wrong?

```
int k, y;
y = k;

char *p, c;
char st[10] = “hello”;
c = *p;                //                , but wrong expression
*p = ‘a’;
p++;
p = st + 1;
st++;                  //
```

[Review] Notices for Using Pointers (2/2)

☑ Initialization of the String

- Ex. What is the difference between these two statements?

```
char s1[] = “hello”;        // Array
char *s2 = “hello”;         //
s1[0] = ‘a’;                //
*s1 = ‘a’;                  //
s2[0] = ‘a’;                //       ! We cannot change the constant.
*s2 = ‘a’;                  //       ! We cannot change the constant.
```

Functions and Structures (1/3)

☑ Characteristics of Functions for Handling Structures

- Structure variables behave like basic, single-valued variables.
- We can pass the structures by value to the functions like ordinary variables.

☑ Two Ways to Pass and Return Structures

- It uses a copied structure, not the original one.
- It uses when the structure is relatively compact.

- It uses an original structure.
- It saves time and memories when the structures is huge.

Functions and Structures (2/3)
- Passing Structure Addresses

☑ Differences between Passing by Value and Address when Calling the Function

- We can pass the address of the structure (&pplace), rather than the structure (pplace) itself.
- We can use the formal parameter as Polar *type pointer(const Polar *pda), instead of Polar type structure(dapos).
- We can use the indirect membership operator (‘->’) rather than the membership operator (‘.’) because the formal parameter is a pointer.

Functions and Structures (3/3)
- Comparison between Passing by Value and Address

Passing Structure

```
struct Polar                // Structure Template
{
    double distance;
    double angle;
}

void show_polar(Polar        );      // Prototype

int main()
{
    Polar pplace;
    …
    show_polar(         );           // Call
    …
}

void show_polar(Polar        )       // Definition
{
    …
    cout << "Distance=" <<                    ;
}
```

Passing Structure

```
struct Polar                // Structure Template
{
    double distance;
    double angle;
}

void show_polar(const Polar        );// Prototype

int main()
{
    Polar pplace;
    …
    show_polar(           );          // Call
    …
}

void show_polar(const Polar        )//Definition
{
    …
    cout << "Distance=" <<                    ;
    …
}
```

Recursion (1/3)

☑ Recursion is simple, but it is very important tool in certain types of programming.

☑ Recursive function includes a statement that calls the function itself.

☑ Ex. n factorial (n!)

$$n! = (n)(n-1)(n-2) \cdots (2)(1)$$

$$n! = \begin{cases} (n)(n-1)! & \text{in case of } n \geq 2 \\ 1 & \text{in case of } n = 1 \end{cases}$$

Recursion (2/3)
- n! Calculation Program

☑ Using Sequential Expression

-
  ```
  int seq_factorial(int n)
  {




  }
  ```

☑ Recursive n! Calculation Program

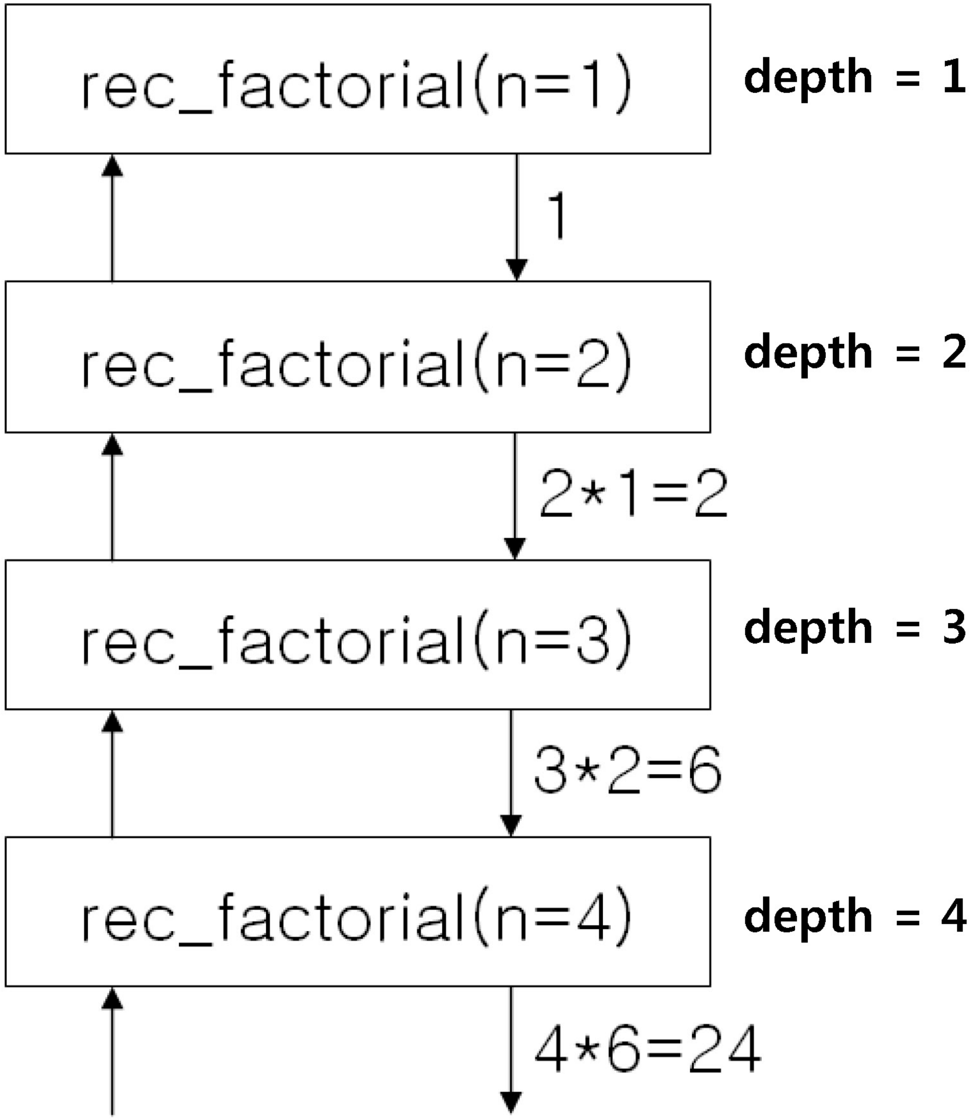
-
  ```
  int rec_factorial(int n)
  {


  }
  ```

Recursion (3/3)
- Execution Procedure of a Recursive n! Calculation Program

☑ A recursive program is more simple than a sequential program, but

.

☑ Use when the depth of recursion is not huge.



Pointers to Functions (1/3)

☑ Pointers to Functions (or Function Pointers)

- Functions also have their addresses. Thus, we can define a function that uses an address of another function as a parameter.

☑ Steps for Using Function Pointers

- Obtain the address of a function.
- Declare a pointer to a function.
- Use a pointer to a function to invoke the function.

Pointers to Functions (2/3)
- Process of Using Function Pointers

☑ Obtaining the Address of a Function

.

- Ex.

```
process(think);        // It passes              to process().
process(think());      // It passes                  to process().
```

☑ Declaring a Pointer to a Function

- Like ordinary pointers, function pointers have to specify to what type of function the pointer points.
- Ex.

```
double gildong(int);   // Function prototype
double *ff(int);       // 'ff()' is a function that returns a pointer.
double (*pf)(int);     // 'pf' is a
```

.

Pointers to Functions (3/3)
- Process of Using Function Pointers (Continued)

☑ Using a Pointer to Invoke a Function

- When we point other functions with function pointers, we have to match the return data type and the function signature.

  : (name doesn't matter)

  - Ex. void print(double d, int width); // Signature is 'double, int'.

- Just use a function pointer to call the function instead of the function name.

  - Ex.

```
double gildong(int);
double (*pf)(int);
pf = gildong;                    // pf points to gildong()
double x = gildong(4)            // call gildong() using the function name
double y = (*pf)(5)              // call gildong() using the pointer pf
```

- Using a function pointer as a parameter of the function

  - Ex.

```
void estimate(int lines, double (*pf)(int));
// The second argument is pointer to a type double function that takes a type int
argument.

estimate(50, gildong);           // 'estimate()' uses 'gildong()'.
```

Summary (1/2)

- ☑ Functions are the C++ programming modules. To use a function, we need to

  .

- ☑ By default, . This means that the formal parameters in the function definition are new variables that are initialized to the values provided by the function call. Thus, C++ functions protect the integrity of the original data by working with copies.

  . Technically, this is still passing by value because the pointer is a copy of the original address, but the function uses the pointer to access the contents of the original array.

Summary (2/2)

- ☑ C++ provides three ways to represent C-style strings: . All are type , so they are passed to a function as a type char* argument.

- ☑ C++ treats structures the same as basic types, meaning that we can pass them by value and use them as function return types.

- ☑ A C++ function can be ; that is, the code for a particular function can include a call of itself.

of a C++ function . By using a function argument that is a pointer to a function, we can pass to a function the name of a second function that we want the first function to evoke.

Practice 1 (1/2)

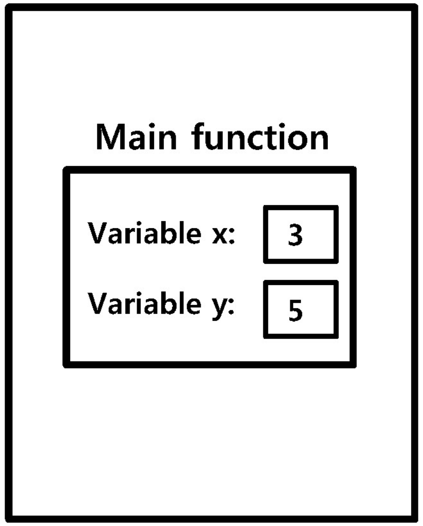
☑ Define a ‘swap’ function which switches two input values.

Get the address of the variable.

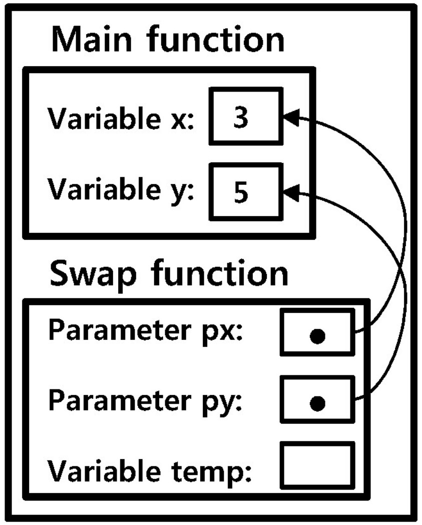
```
void main()
{
        int x = 3, y = 5;          // (1)
        swap(&x, &y);              // (2)(6)
}

void swap(int *px, int *py)        // (2)
{
        int temp;
        temp = *px;                // (3)
        *px = *py;                 // (4)
        *py = temp;                // (5)
}
```

Practice 1 (2/2)

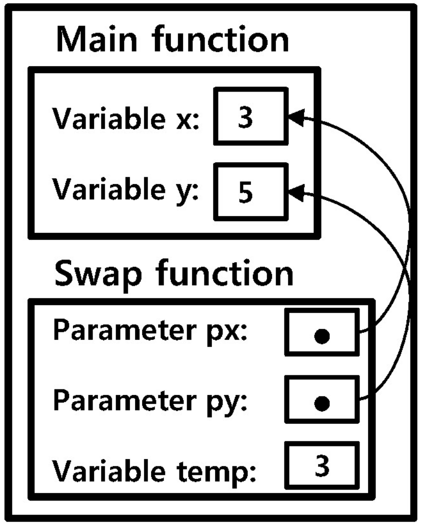
```
void main()
{
        int x = 3, y = 5;           // (1)
        swap(&x, &y);               // (2)(6)
}

void swap(int *px, int *py)         // (2)
{
        int temp;
        temp = *px;                 // (3)
        *px = *py;                  // (4)
        *py = temp;                 // (5)
}
```



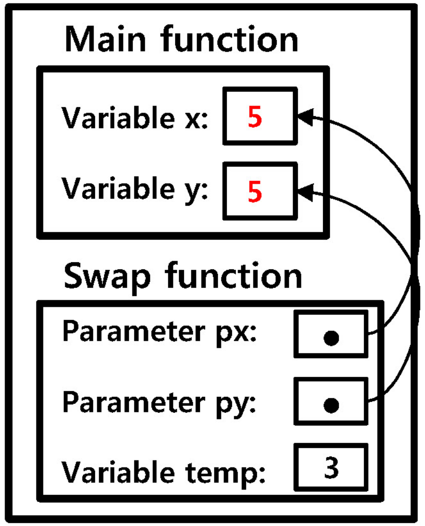
(1)



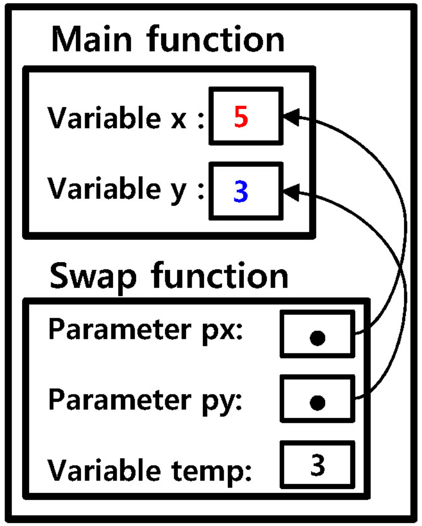
(2)



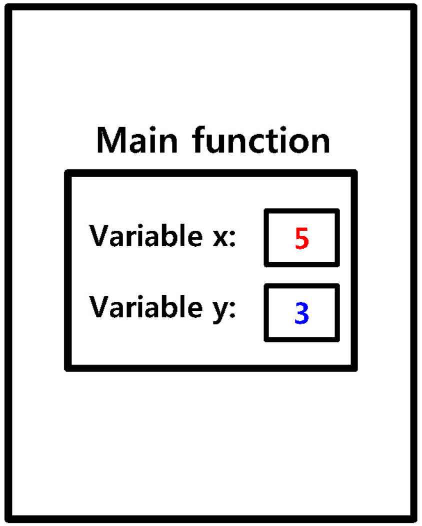
(3)



(4)



(5)



(6)

Practice 2

☑ Define a function that reads an array and its size, and calculates the average of the value in an array.

```
#include <fstream>                // Header for file input & output
float average(int, float[]);      // Size of the array, array

void main()
{
    ifstream fin;                 // Declare input file identifier ‘fin’.
    fin.open(“score.txt”);        // Open input file ‘score.txt’.
    fin >> n;                     // Read the number of classes ‘n’ in the input file.
    for (int i = 0; i < n; i++) {
        fin >> np;                // Read the number of students in a class ‘np’
                                     in the input file.
        for (int j = 0; j < np; j++) {
            fin >> score[j];      // Read the j-th student’s score in the input file.
        }
        Call a average function, and store a return value at avg[i].
    }
    fin.close();                  // Close the input file.
    Output the average score of each class, avg[i]
}

Define average function
```

Practice 3

- ☑ Make a program with Defining functions described as below and calling them.
  - ■ Get a string and return n characters from the right.
    char * right(char *s, int n);
  - ■ Get a string and return n characters from the left.
    char *left(char *s, int n);
  - ■ Get a string and return n characters from the m-th character.
    char *mid(char *s, int m, int n);

Practice 4

☑ The Fibonacci sequence is like as below;

0, 1, 1, 2, 3, 5, 8, 13, 21, 31, 51, …

☑ By definition, the first two numbers in the Fibonacci sequence are 1 and 1, or 0 and 1, and each subsequent number is the sum of the previous two. In mathematical terms Fibonacci numbers is defined like as below;

F0 = 1 // If n = 0
F1 = 1 // If n = 1
$F_n = F_{n-1} + F_{n-2}$ // If n > 1

☑ Make a program that calculates the Fibonacci sequence.

Practice 5

☑ Make a vector program.

- ■ Declare a vector with a structure.
- ■ Define a function that calculates dot (scalar) product and cross (vector) product, and make a program that calls the function you have defined.