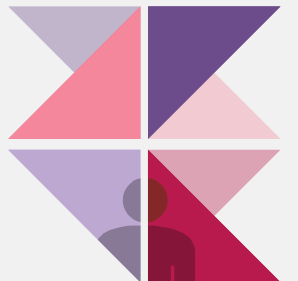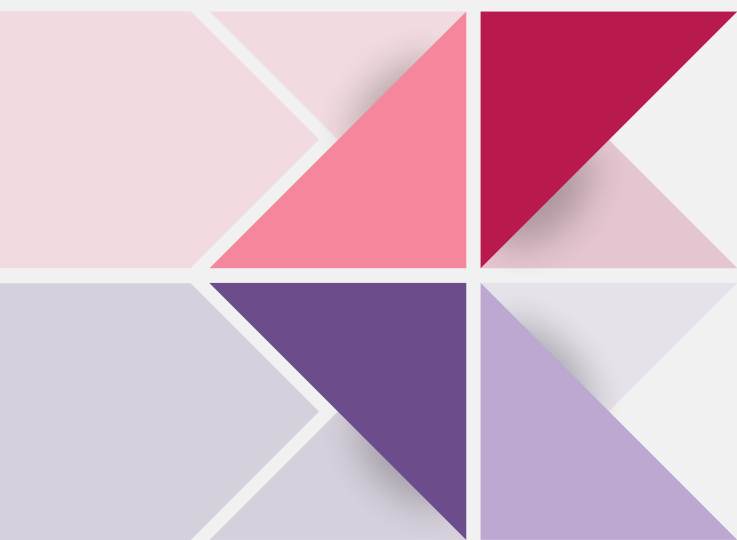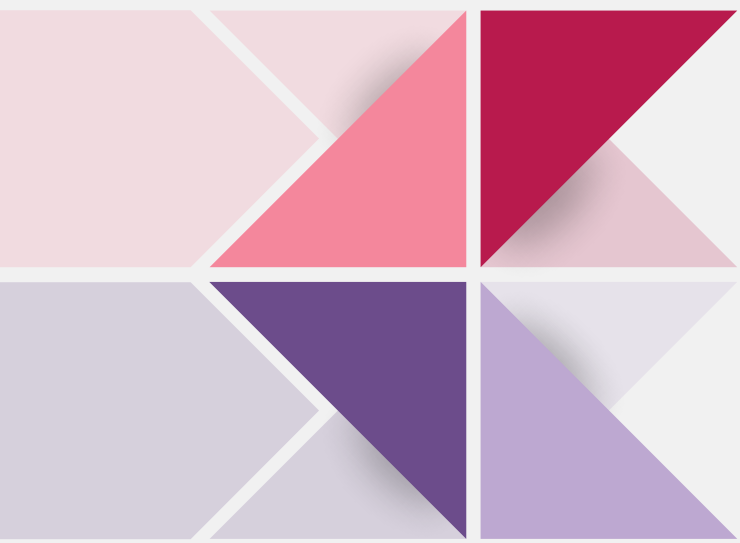# Supplementary

# 00
# Midterm

# Supplementary

Midterm

Date and Place

- ➢ 2022 / 11 / 10 (Thursday) 13:10-16:05
- ➢ 教學一館 33303
- ➢ You will not take the test if arriving the 33303 after 13:35

# 01
# Printf

# Supplementary
Printf

## Output - printf()

➢ The printf function must be supplied with a format string, followed by any values that are to be inserted into the string during printing
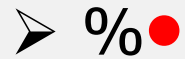
printf(*string*, *expr* 1, *expr* 2, ...);

➢ The format string may contain both ordinary characters and conversion **specifications**, which begin with the % character

```c
int p = 3;
printf("The value is %d\n", p);
```

# Supplementary
## Printf

## Conversion specifications

➢ %●

- %c - Character
- %d - Integer
- %f - Floating point
- %s - String
- %e - Exponential format
- %g - Either exponential format or fixed decimal format, depending on the number's size

```
float i = 123.223;
printf("i = %f\n", i);
```

```
float i = 123.223;
printf("i = %e\n", i);
```

```
float i = 123.223;
printf("i = %g\n", i);
```

```
i = 123.223000
```

```
i = 1.232230e+002
```

```
i = 123.223
```

# Supplementary
## Printf

## Output - printf()

➢ A conversion specification can have the form %m.pX or %-m.pX, where the m and p are integer constants and X is a letter

    printf("%10.2f\n", i);   // m = 10, p = 2, X = f

    printf("%10f\n", i);   // m = 10, p = missing, X = f

    printf("%f\n", i);   // m and p are miss, X = f

➢ m specifies the minimum number of characters to print

    int i = 123;
    printf("%10d\n", i);   // It will print ● ● ● ● ● ● ● 123

➢ Putting a minus sign in front of m causes left justification

    printf("%-10d\n", i);   // It will print 123 ● ● ● ● ● ● ●

# Output - printf()

➢ The meaning of the precision, p, depends on the choice of X, the conversion specifier

➢ The **d** specifier is used to display an integer in decimal form

 ➢ p indicates the minimum number of digits to display (extra zeros are added to the beginning of the number if necessary)
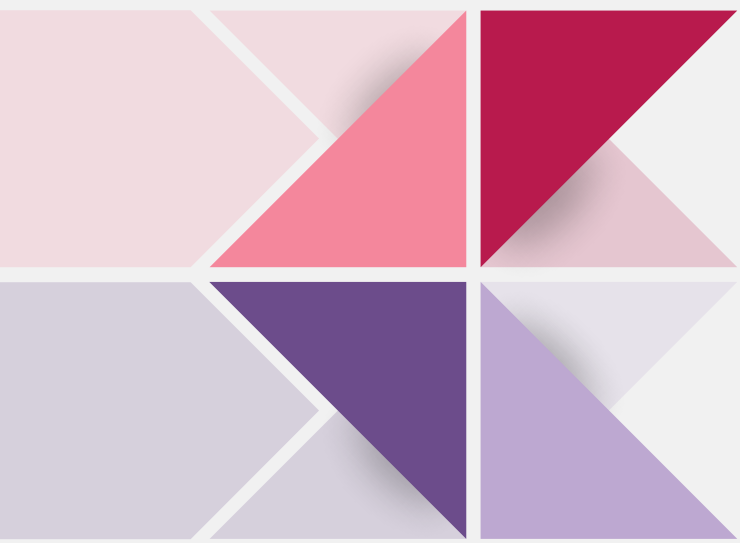
```
int i = 314;
printf("i = %.4d\n", i);
```
```
i = 0314
```

```
float i = 3.141592;
printf("i = %.4f\n", i);
```
```
i = 3.1416
```

```
float i = 3.141592;
printf("i = %10.4f\n", i);
```
```
i =     3.1416
```

8

# 02
## ++ and --

# Supplementary

++ and --

Increment and decrement operators

- ➤ "++" and "--"
    - • ++ : adds 1 to its operand
    - • -- : subtracts 1 to its operand
- ➤ They can be employed as prefix (++i) or postfix (i++) operators
- ➤ They have side effects

    ```
    int prefix_i = 1;                              int postfix_i = 1;
    printf("prefix_i is %d\n", ++prefix_i);  printf("postfix_i is %d\n", postfix_i++);
    printf("prefix_i is %d\n", prefix_i);      printf("postfix_i is %d\n", postfix_i);
    ```

- ➤ "++prefix_i" means "increment prefix_i immediately", while "postfix_i++" means "use the old value of postfix_i for now, but increment it later"
- ➤ How much later? The C standard doesn't specify a precise time, but it's safe to assume that the variable will be incremented before the next statement is executed

# Supplementary
## ++ and --

x = y += z++-i+--j / -k

x = y += (z++)-i+--j / -k

x = y += (z++)-i+(--j) / -k

x = y += (z++)-i+(--j) / (-k)

x = y += (z++)-i+((--j) / (-k))

x = y += ((z++)-i)+((--j) / (-k))

x = y += (((z++)-i)+((--j) / (-k)))

x = (y += (((z++)-i)+((--j) / (-k))))

| Precedence | Name | Symbol(s) | | |
|------------|------|-----------|---|---|
| 1 | Postfix increment | Operand++ | | |
| | Postfix decrement | Operand-- | | |
| 2 | Prefix increment | ++Operand | | |
| | Prefix decrement | --Operand | | |
| | Unary plus | +Operand | | |
| | Unary minus | -Operand | | |
| 3 | Multiplicative | Operand * / % Operand | | |
| 4 | Additive | Operand + - Operand | | |
| 5 | Assignment | Operand | = *= /= %= += -= | Operand |

# Supplementary

++ and --

(a)
i = 1;
printf("%d ", i++ - 1);
printf("%d", i);

(b)
i = 10, j = 5;
printf("%d ", i++ - ++j);
printf("%d %d", i, j);

(c)
i = 7, j = 8;
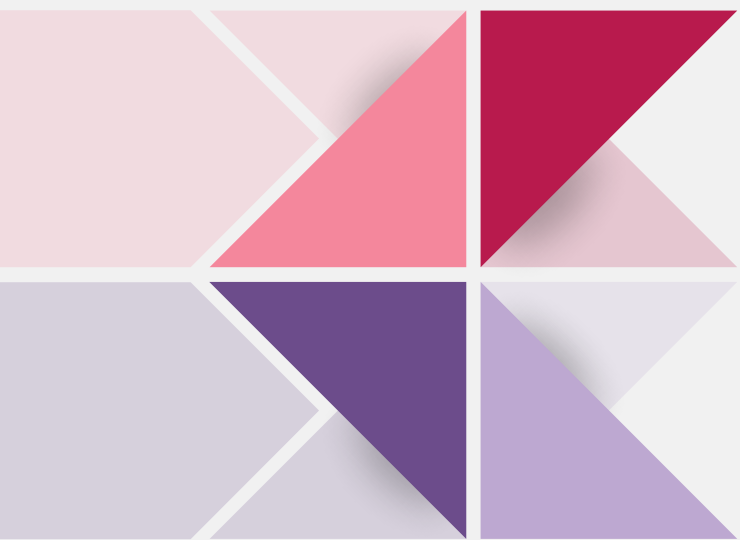printf("%d ", i++ - --j);
printf("%d %d", i, j);

(a)
0 2

(b)
4 11 6

(c)
0 8 7

# 03
## If and Switch

if (*expression*) *statement* else if *statement* else *statement*

```
if ( m < n)
    printf("m is less than n\n");
else if (m == n)
    printf("m is equal to n\n");
else
    printf("m is greater than n\n");
```

*expression* 1 *? expression* 2 : *expression* 3

```
int x=1, y=2, z;
if (x > y) z = x;
else z = y;
if (x > 0) z= x + y;
else z = 0 + y;
```

→

```
int x=1, y=2, z;
z = x>y? x : y;
z = (x >=0? x : 0) + y;
```

14

# Supplementary

If and Switch

```
switch (grade)
{
    case 3:
        printf("Very good\n");
        break;
    case 2:
        printf("Good\n");
        break;
    case 1:
        printf("Average\n");
        break;
    case 0:
        printf("Failing\n");
        break;
    default:
        printf("Illegal grade\n");
        break;
}
```

```
switch (grade)
{
    case 3:
        printf("Very good\n");
    case 2:
        printf("Good\n");
    case 1:
        printf("Average\n");
    case 0:
        printf("Failing\n");
    default:
        printf("Illegal grade\n");
}
```

```
switch (grade)
{
    case 3:
    case 2:
    case 1:
        printf("Passing\n");
        break;
    case 0:
        printf("Failing\n");
        break;
    default:
        printf("Illegal grade\n");
        break;
}
```
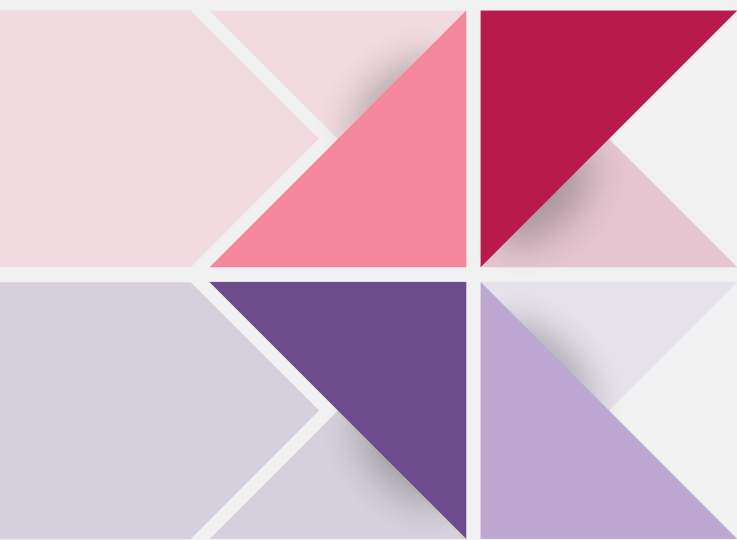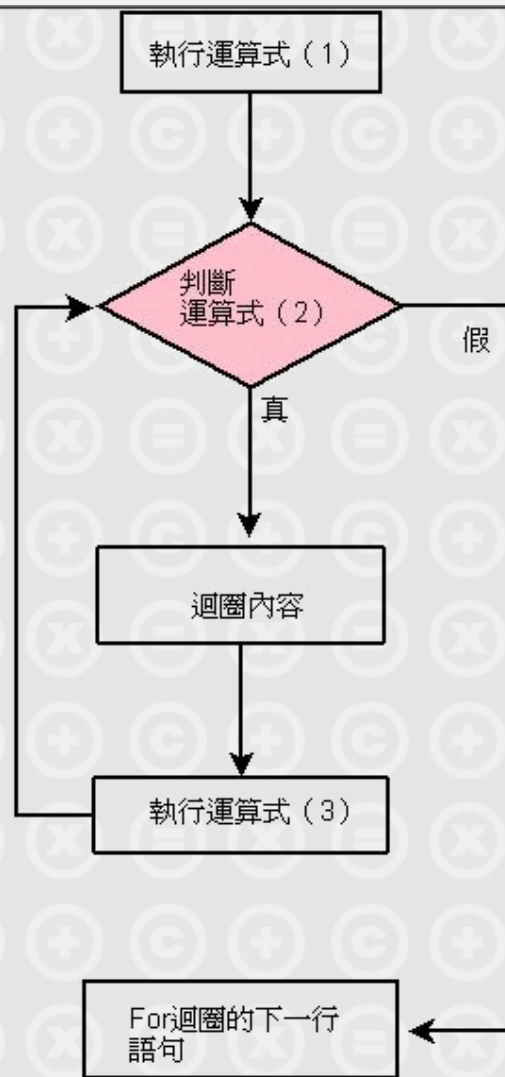
# 04
# For Loop
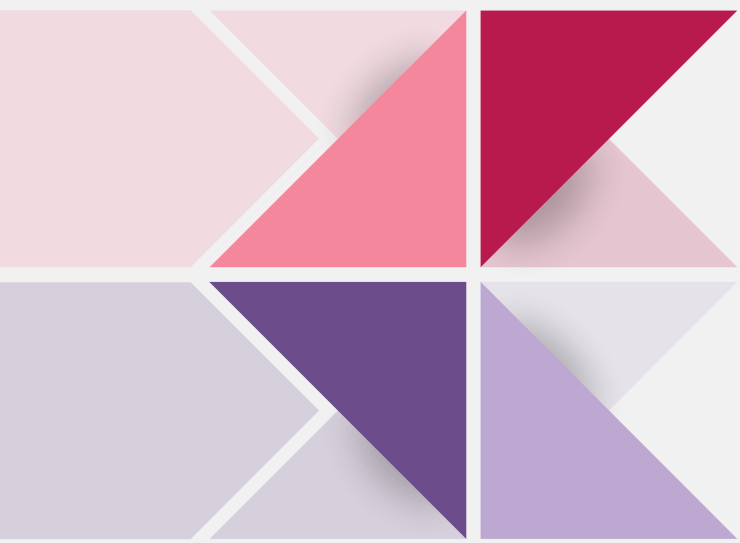
# Supplementary
## For Loop

*for* (*exp* 1; *exp* 2; *exp* 3)
{

     *statements*

}

1) 先執行運算式1。

2) 再執行運算式2，若其值為真（非0），則執行for語句中指定的內嵌語句，然後執行下面 第 3 步；若其值為假（為0），則結束迴圈，轉到第 5 步。

3) 執行運算式 3。

4) 轉回上面第 2 步繼續執行。

5) 迴圈結束，執行 for 語句下面的一個語句。

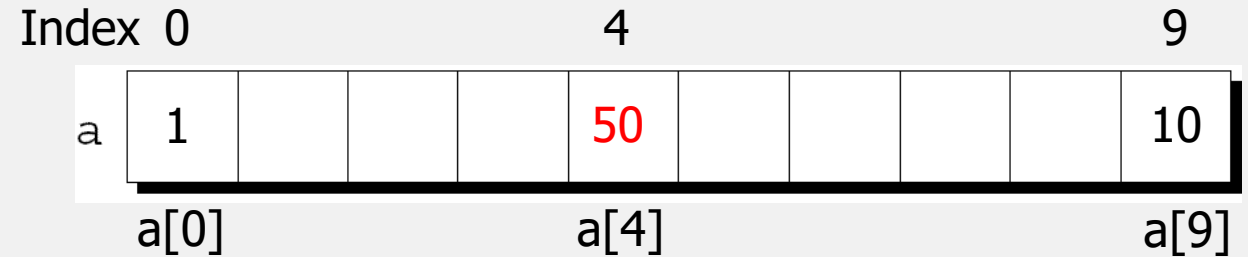其執行過程可用右圖表示。

Reference

# 05
# Array

# Supplementary

Array

How to access elements in the array?

➢ Write the array name followed by an integer value in square bracket

- This is referred to as subscripting or indexing the array
- Important concept: the range of index is from 0 to N-1

```
int a[10];
a[4] = 50;
```

Index 0                        4                     9

a   | 1 | | | | 50 | | | | | 10 |

     a[0]                 a[4]                a[9]

```
int a[10], i;

for (i = 0; i < 10; i++)
    a[i] = 0;
```

# Supplementary

Array

An array, like any other variable, can be given an initial value at the time it's declared

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

int a[10] = {1, 2, 3, 4}; //initial value is {1, 2, 3, 4, 0, 0, 0, 0, 0, 0}

int a[10] = {0};          //initial value is {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}

int a[10] = {6}; -> ?     //initial value is {6, 0, 0, 0, 0, 0, 0, 0, 0, 0}

int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```
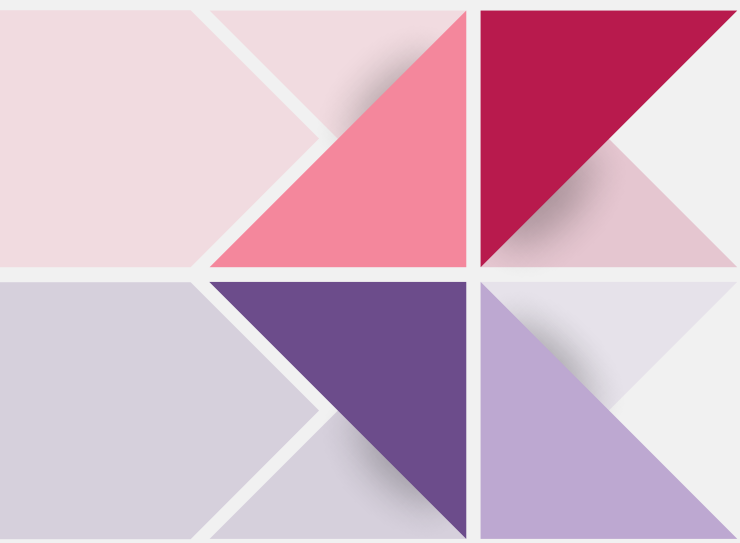
# Supplementary

Array

We can create an initializer for a two-dimensional array by nesting one-dimensional initializers

```
int x[3][4] = {{1, 1, 1, 1},
               {0, 1, 0, 1},
               {1, 0, 1, 0}};
```
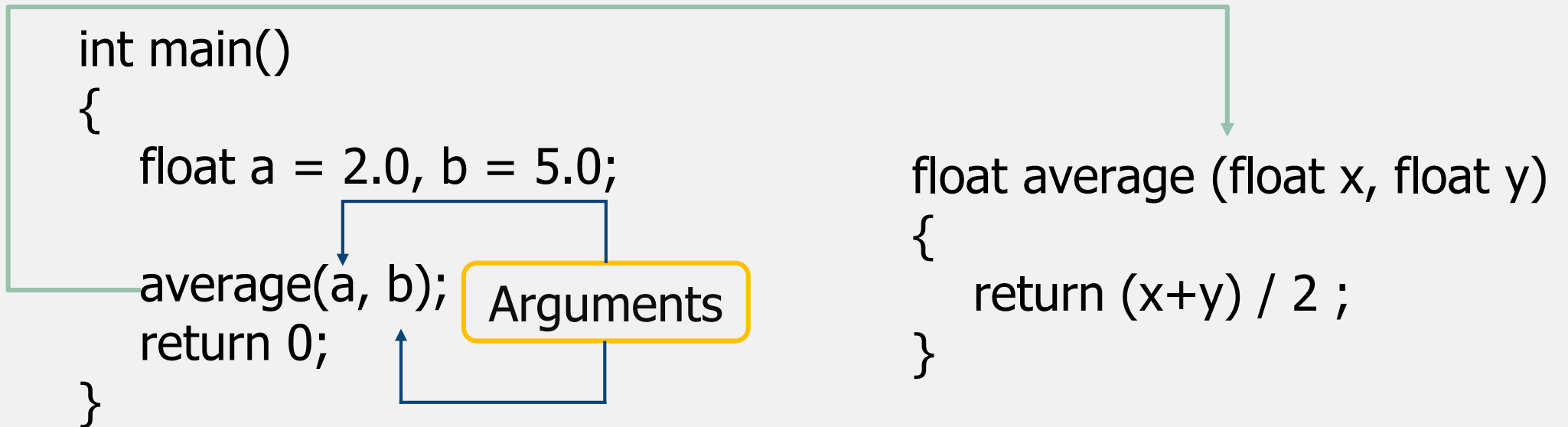
# 06
# Function

# Supplementary
Function

A function is called with the combination of <span style="color:red">function name</span> and a list of <span style="color:red">arguments</span>

```
int main()
{
    float a = 2.0, b = 5.0;

    average(a, b);      Arguments
    return 0;
}

float average (float x, float y)
{
    return (x+y) / 2 ;
}
```

Arguments are used to provide information to a function

➤ If calling average(a, b), the values of *a* and *b* will be <span style="color:red">*copied*</span> into the parameters x and y

# Supplementary

Function

If calling a function before defining it, error messages will be shown by the compiler

➢ The compiler will assume that average returns an int value

```
int main()
{
    float a = 2.0, b = 5.0;
    printf("Average is: %f\n", average(a, b));
    return 0;
}

float average (float x, float y)
{
    return (x+y) / 2 ;
}
```

```
test.c: In function 'main':
test.c:5:29: warning: implicit declaration of function 'average' [-Wimplicit-function-declaration]
        printf("Average: %f\n", average(2,3));
                                ^~~~~~~
test.c: At top level:
test.c:9:7: error: conflicting types for 'average'
 float average(int x, int y)
       ^~~~~~~
test.c:5:29: note: previous implicit declaration of 'average' was here
        printf("Average: %f\n", average(2,3));
                                ^~~~~~~
```

# Supplementary

Function

## How to avoid?

- ➤ Declare a function before calling it to provide the compiler with a brief glimpse
- ➤ The declaration must be consistent with the function's definition
- ➤ General form of function declaration

(*return type*) *function_name* (*parameters*);

float average (float x, float y);

```
int main()
{
    float a = 2.0, b = 5.0;
    printf("Average is: %f\n", average(a, b));
    return 0;
}
```

# Supplementary
Function

## Array arguments

➢ When a function parameter is a one-dimensional array, the length of the array can be left unspecified such as

```
int f(int a[])  // no length specified
{
    ...
}
```

➢ However, there is no any easy way in C for a function to determine the length of an array passed to it

```
int f(int a[])
{
    printf("sizeof(a) = %d\t sizeof(a[0]) = %d", sizeof(a), sizeof(a[0]));
    return sizeof(a) / sizeof(a[0]);
}
```

sizeof(a) = 4    sizeof(a[0]) = 4

# **Supplementary**
Function

## How recursion works?

```
x = fact(3);

int fact(int n)
{
    if (n <= 1)
        return 1;
    else
        return n * fact(n - 1);
}
```

fact(3) finds that 3 is not less than or equal to 1, and it calls fact(2)

fact(2) finds that 2 is not less than or equal to 1, and it calls fact(1)

fact(1) finds that 1 is less than or equal to 1, and it returns 1,

causing fact(2) to return 2 × 1 = 2, and

causing fact(3) to return 3 × 2 = 6