Introduction to C Programming

Lecture 5: functions

Wenjin Wang
wangwj3@sustech.edu.cn

10-9-2022

Course syllabus

Nr.	Lecture	Date
1	Introduction	2022.9.9
2	Basics	2022.9.16
3	Decision and looping	2022.9.23
4	Array & string	2022.9.30
5	Functions	2022.10.9 (补)
6	Pointer	2022.10.14
7	Self-defined types	2022.10.21
8	I/O	2022.10.28

Nr.	Lecture	Date
9	Head files & pre-processors	2022.11.4
10	Review of lectures	2022.11.11
11	Soul of programming: Algorithms I	2022.11.25
12	Soul of programming: Algorithms II	2022.12.2
13	R&D project	2022.12.9
14	R&D project	2022.12.16
15	R&D project	2022.12.23
16	Summary	2023.12.30

Recap last lectures

- To enable group processing of data, we can use **array** data structure
- Array has **fixed size** and can only hold the data with the **same type**
- Different **types** of array can be created, e.g. int a[6], float f[10], char c[3]
- 1D char array is **string**!
- Different **dimensional** arrays can be created, e.g. int a[3][5] means 3 rows and 5 columns.
- Array enables the processing of **vectors, matrices and strings**.
- **Sorting** is an important function for array operation (3 types of sorting).

Recap last lecture

Declare variable

```
int a;
float f;
char c;
```

Declare array

```
int a[10];
float f[20];
char c[5];
```

Recap last lecture

Declare & init. variable

```
int a = 10;
float f = 3.14;
char c = ‘a’;
```

Declare & init. array

```
int a[] = {1, 2, 3};
float f[] = {1.1, 5.3};
char c[] = “Hello”;
```

Recap last lecture

Declare and initialize an int array (separately):

- int a[10]; **// declare**
- a[0] = 3, a[1] = 2, …., a[9] = 7; **// initialize**

Declare and initialize an int array (jointly):

- int a[10] = {3, 2, 1, 5, 6, 8, 9, 2, 0, 7}; **// declare and initialize**
- int a[] = {3, 2, 1, 5, 6, 8, 9, 2, 0, 7}; **// declare and initialize**

Access the element in an array:

- printf(“a[5] = %d”, a[5]); **// access the array**

Recap last lecture

int a[10]; // array length is 10

a[6] = 5; // 7^{th} element is 5

a[10] = 1; // not allowed!!!

Recap last lecture

Arithmetic operations can be applied N-D arrays

1D array

element-wise

+, -, .*, ./

sorting

bubble, selection, insertion

2D matrix

element-wise

+, -, .*, ./

cross

*, /

Recap last lecture

C-defined string operations can be applied

Operators	Description	Example s1=A, S2 = B;
strcpy(s1, s2)	Copy s2 into s1	s1 = B
strcat(s1, s2)	Concatenate s1 and s2	S1 = AB
strlen(s1)	Return length of s2	Length = 1
strcmp(s1, s2)	Compare s1 and s2	A<B, return -1
strlwr(s1)	Convert s1 to lower case	A to a
strupr(s1)	Convert s1 to upper case	A to A

Objective of this lecture

You can write your own functions in C!

Content

1. Declare/define/call a function
2. Variable scope
3. Recursion

Content

1. **Declare/define/call a function**
2. Variable scope
3. Recursion

Multiple functions in life

A practical task usually has many functions!

Identification at “深圳北站”


```
int main()
{
    // face detection
    // face mask detection
    // face recognition
    // decision making
    return 0;
}
```

```
detectFace()
{…}
```

```
detectMask()
{…}
```

```
recogFace()
{…}
```

```
makeDec()
{…}
```

Multiple functions in life


```
int main()
{
    // select a KFC
    // Whether it is sold out ?
    // how much ?
    // pay
    return 0;
}
```

```
positioning()
{…}
```

```
Check_inventory()
{…}
```

```
Cal_price()
{…}
```

```
Wechat_pay()
{…}
```

Multiple functions in life

```
int main()
{
    // Do you have enough mana?
    // select a target
    /* Whether the target is
       eliminated ? */
    return 0;
}
```

```
check_mana()
{…}
```

```
aim()
{…}
```

```
cal_damage()
{…}
```

Function

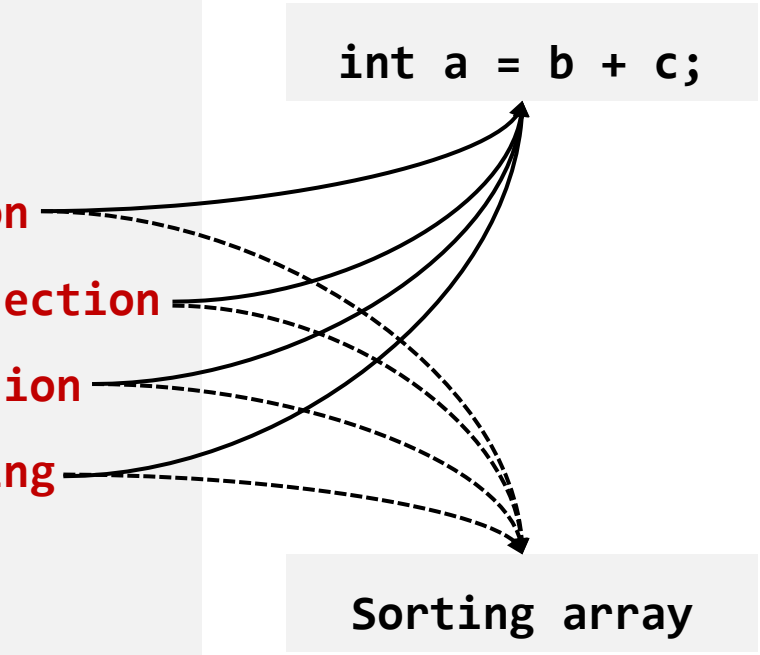
Main is a function, performing a task!

```
int main()
{
    // do nothing or do something!!!
    return 0;
}
```

Function

Main usually includes multiple tasks, preferred not to write all tasks in a big main!!!

```
int main()
{
    // face detection
    // face mask detection
    // face recognition
    // decision making
    return 0;
}
```



- Not modular
- Difficult to maintain
- Difficult to hand-over
- Cannot be re-used

Function

Make functions independent of the main!

```
int detectFace()
{  // do something
    return 0;
}
```

```
int main()
{
    // face detection
    // face mask detection
    // face recognition
    // decision making
    return 0;
}
```

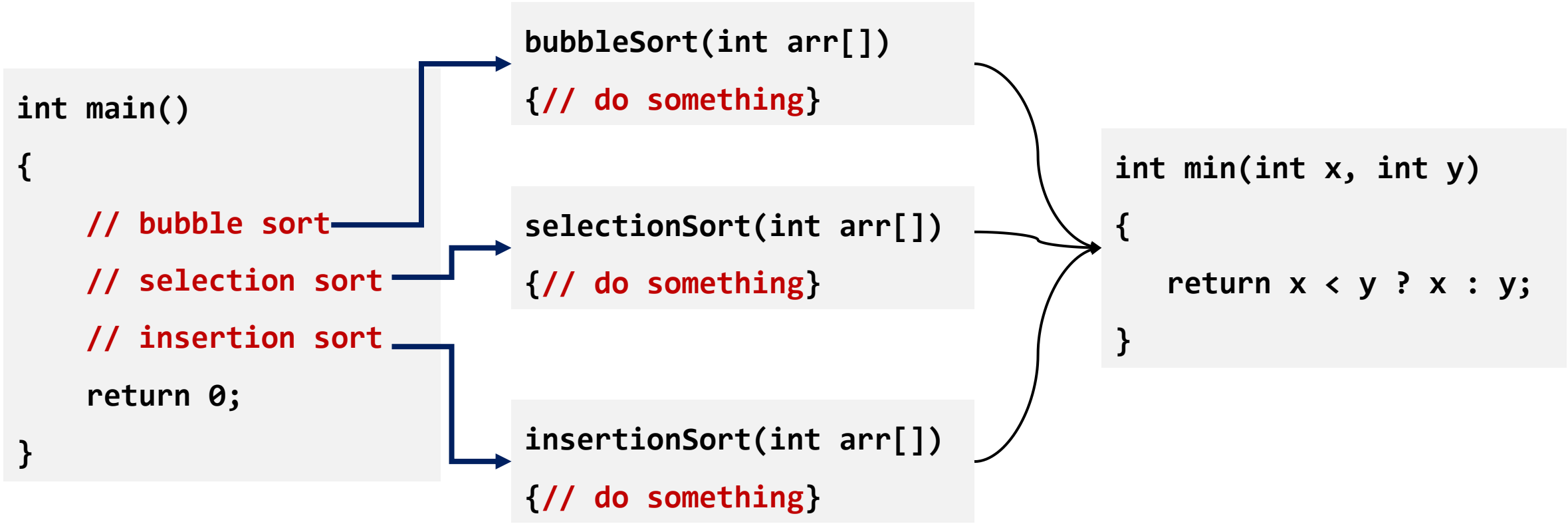
```
int recogFace()
{  // do something
    return 0;
}
```

```
int detectMask()
{  // do something
    return 0;
}
```

```
int makedecision()
{  // do something
    return 0;
}
```

Function

Make functions independent of the main!



Function

Function is a group of statements that together perform a task. C provides numerous built-in functions, we can also define our own functions.

```
return_type function_name(parameters)
{
    body of the function
     return;
}
```

```
int detectFace()
{  // do something
    return 0;
}
```

```
float detectMask()
{  // do something
    return 100.0;
}
```

```
void recogFace()
{  // do something
}
```

```
char makeDecision()
{  // do something
    return ‘y’;
}
```

C-defined functions

sqrt function in math.h

```
float sqrt(float number)
{
long i;
float x2, y;
const float threehalfs = 1.5F;
x2 = number * 0.5F;
y = number;
i = *(long*)&y;
i = 0x5f3759df - (i >> 1); y = *(float*)&i;
y = y * (threehalfs - (x2 * y * y));
#ifndef Q3_VM
#ifdef __linux__
assert(!isnan(y)); #endif
#endif
return y;
}
```

printf function in stdio.h

```
int printf(const char* fmt, ...)
{
int i;
char buf[256];

va_list arg = (va_list)((char*)(&fmt) + 4);
i = vsprintf(buf, fmt, arg);
write(buf, i);

return i;
}
```

Self-defined function

Declare a function (声明)

Define a function (定义)

Call a function (调用)

How to separate functions

```c
#include<stdio.h>


main()
{
    int x = 20, y = 10;
    int z = x + y;
    int max = x > y ? x:y;
}
```

```c
int sum(int x, int y)
{
    int z = x + y;
    return z;
}
```

```c
int max(int x, int y)
{
    int z = x > y ? x : y;
    return z;
}
```

How to separate functions

```c
#include<stdio.h>


main()
{
    int x = 20, y = 10;
    int z = x + y;
    int max = x>y ? x:y;
}
```

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

You can directly return the result!

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

Function

```
#include<stdio.h>

int sum(int x, int y);

main()
{
    int x = 20, y = 10;
    int z = sum(x, y);
}

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

① Declare function

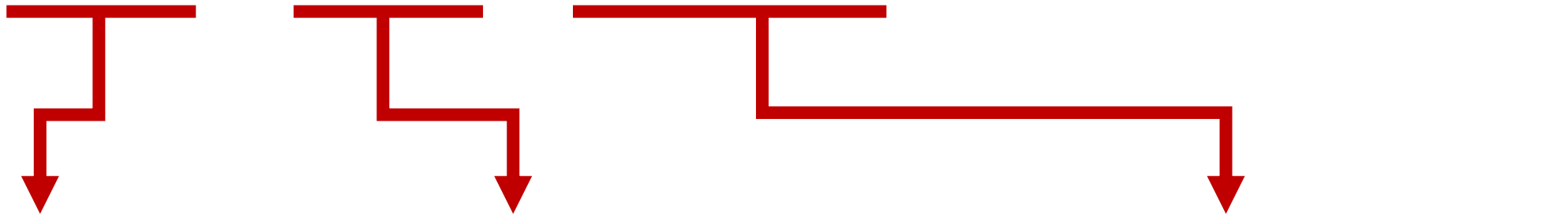
③ Call function

② Define function

```
#include<stdio.h>

int max(int x, int y);

main()
{
    int x = 20, y = 10;
    int z = max(x, y);
}

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

Declare a function

```
int sum(int x, int y);
```



Return type

float, double, char, void

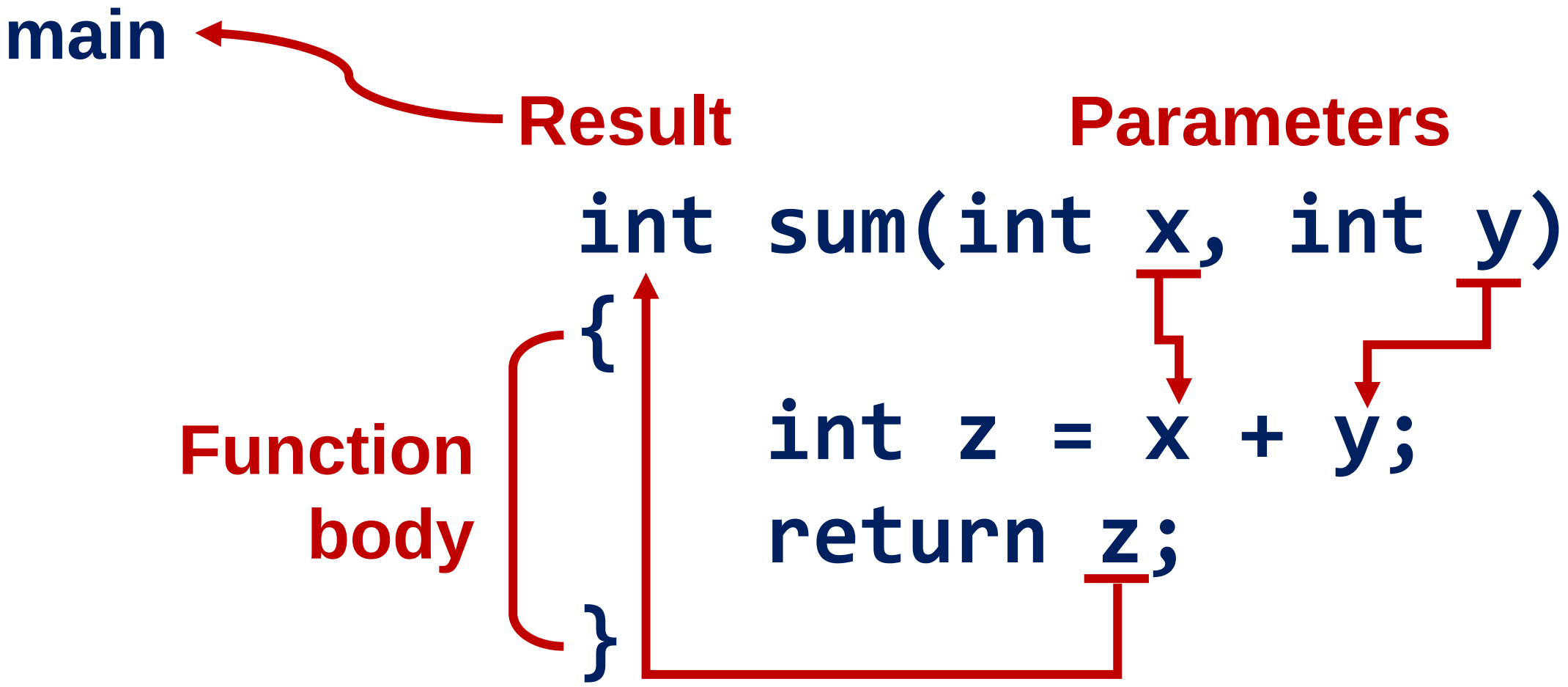
Function name

Any name you like

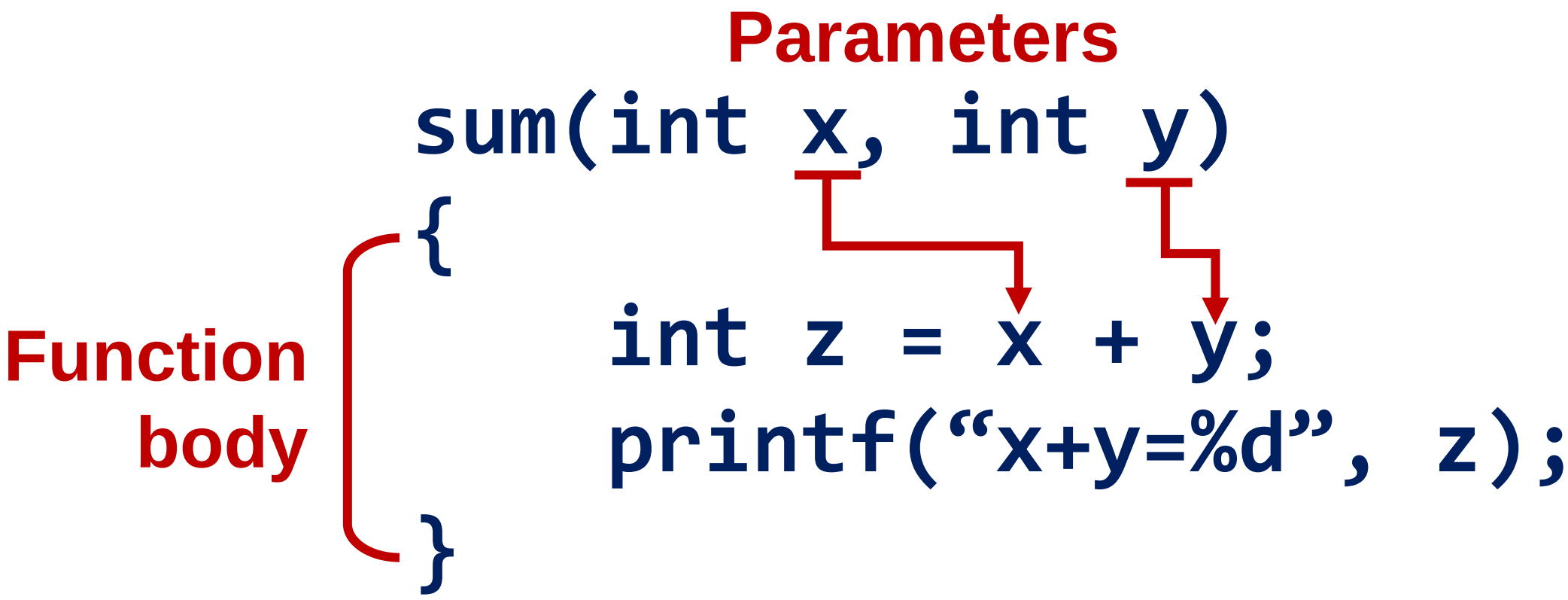
Input parameters（形式参数）

Variables, arrays

Define a function



Define a function



Call a function

```
main()
{
    int x = 20, y = 10;
    int z = sum(x, y);
}
```

Arguments
（实际参数）

Function positioning matters

```c
#include<stdio.h>

int sum(int x, int y)
{
   return x + y;
}

main()
{
   int x = 20, y = 10;
   int z = sum(x, y);
}
```

① Declare and define function before main!!!

② Call function

```c
#include<stdio.h>

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

main()
{
   int x = 20, y = 10;
   int z = max(x, y);
}
```

Function positioning matters

```c
#include<stdio.h>


main()
{
    int x = 20, y = 10;
    int z = sum(x, y);
}


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

Wrong!

Compiler cannot recognize the function declared after main

```c
#include<stdio.h>


main()
{
    int x = 20, y = 10;
    int z = max(x, y);
}


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

Definition must be consistent

```
int sum(int x, int y); // declaration
void sum(int x, int y); // return type matters
int sum(int x, int y, int z); // parameter matters
int sum2(int x, int y); // name matters


int sum(int x, int y) // definition
{return x + y;}
```

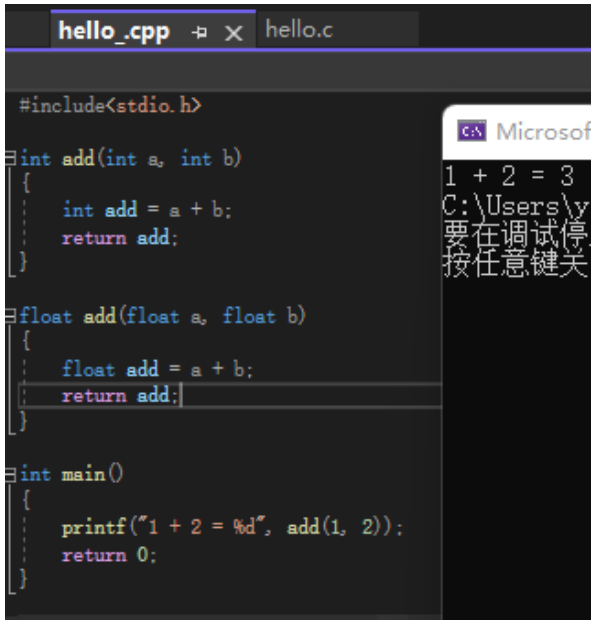
Definition must be consistent

```c
#include<stdio.h>

int add(int a, int b)
{
int add = a + b;
return add;
}

float add(float a, float b)
{
float add = a + b;
return add;
}

int main()
{
printf("1 + 2 = %d", add(1, 2));
return 0;
}
```

If we run this code in C & C++, what will happen?

C

C++



Strongly suggested structure!

```
#include<stdio.h>

int sum(int x, int y);

main()
{
   int x = 20, y = 10;
   int z = sum(x, y);
}

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

①

Declare function

(prompt to read)

③

Call function

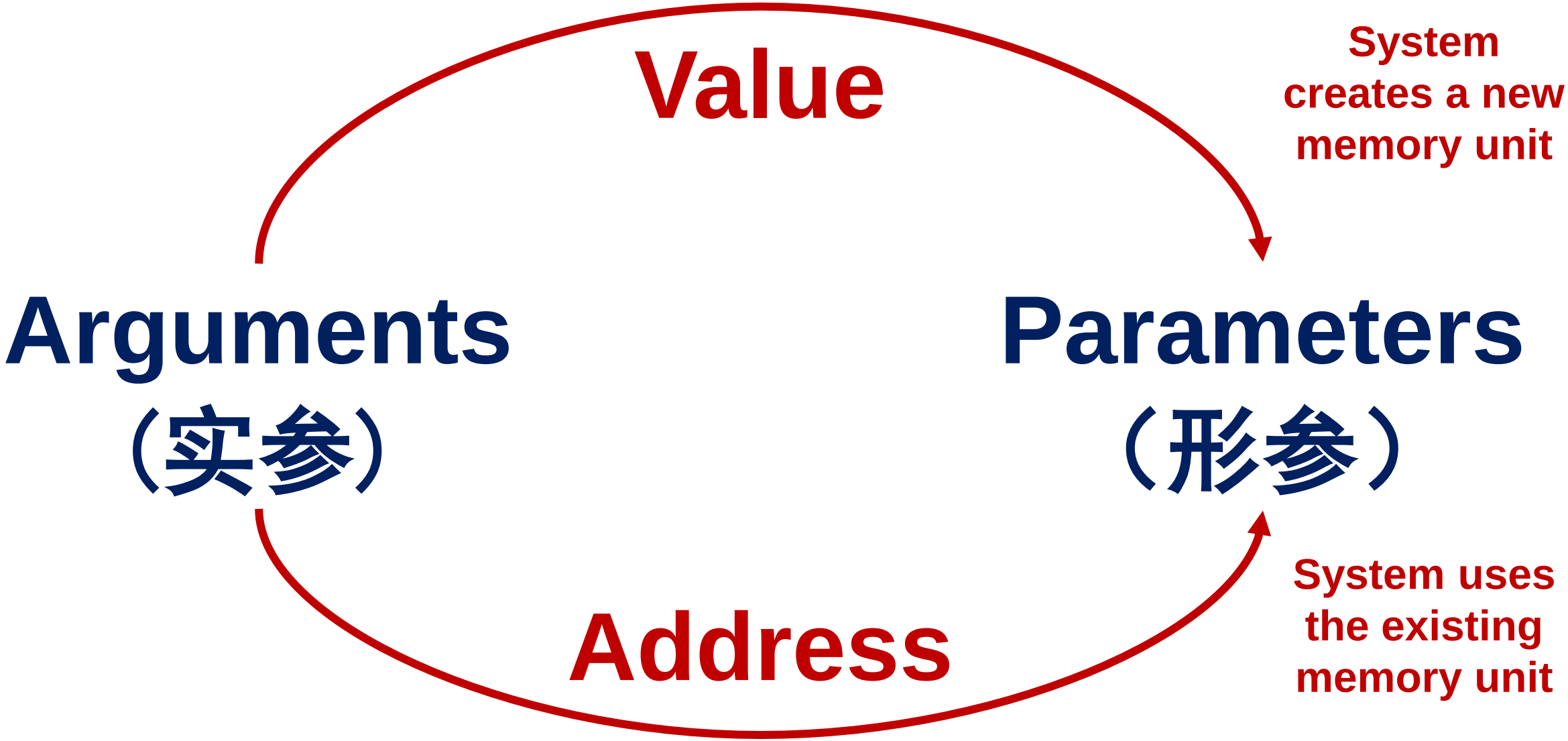
②

Define function

(details of implementation)

```
#include<stdio.h>

int max(int x, int y);

main()
{
   int x = 20, y = 10;
   int z = max(x, y);
}

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

Arguments and parameters



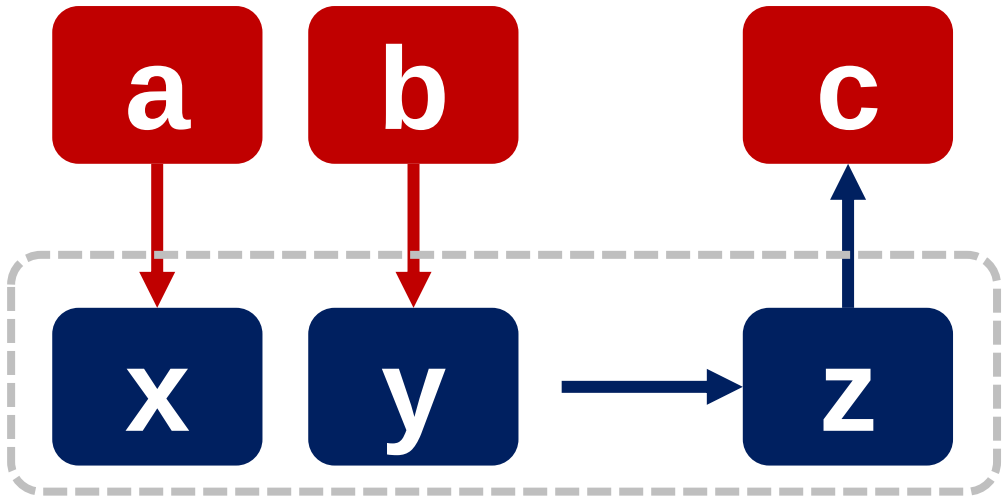
Arguments and parameters

```
#include <stdio.h>

int sum(int x, int y);

main ()
{
     int a = 100;
     int b = 200;

     int c = sum(a, b);
}
```

a and b are arguments
（实参）

x and y are parameters（形参）

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



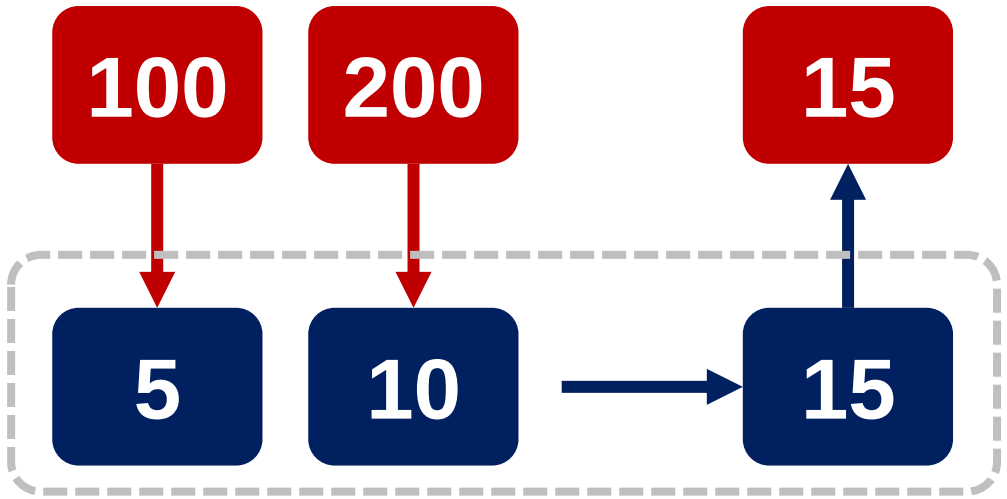
Arguments and parameters

```
#include <stdio.h>

int sum(int x, int y);

main ()
{
     int a = 100;
     int b = 200;

     int c = sum(a, b);
}
```

a and b are arguments
（实参）

x and y are parameters（形参）

```
int sum(int x, int y)
{
     x = 5, y = 10;
     int z = x + y;
     return z;
}
```



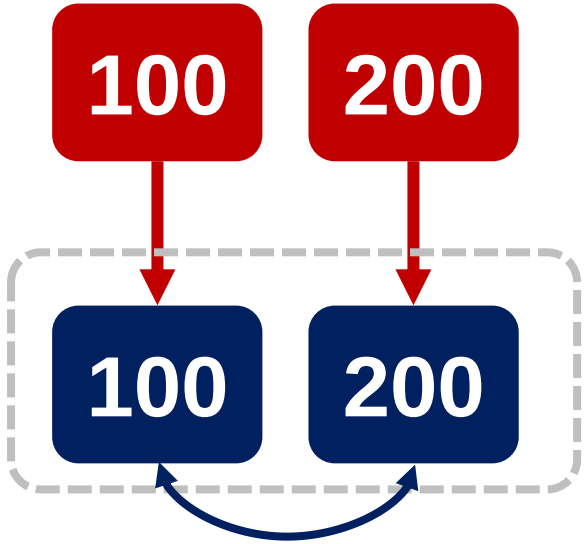
Arguments and parameters

```
#include <stdio.h>

void swap(int x, int y);

main ()
{
     int a = 100;
     int b = 200;

     swap(a, b);
}
```

a and b are arguments
（实参）

x and y are parameters（形参）

```
void swap(int x, int y)
{
     int temp = x;
     x = y;
     y = temp;
}
```



Can a and b be swapped?

Arguments and parameters

```
#include <stdio.h>

void swap(int *x, int *y);

main ()
{
    int a = 100;
    int b = 200;

    swap(&a, &b);
}
```

&a and &b are address of arguments

Pointers

```
void swap(int *x, int *y)
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
```

Can a and b be swapped now?

Arguments and parameters

```
void print_array(int* array,int size)
{
for (int i = 0; i < size; i++)
printf("%d ",array[i]);
return;
}

int add(int a, int b)
{
int add = a + b;
return add;
}
int main()
{
int a[5] = { 0,1,2,3,4 };
print_array(a,5);

add(a[2],a[3]);
return 0;
}
```

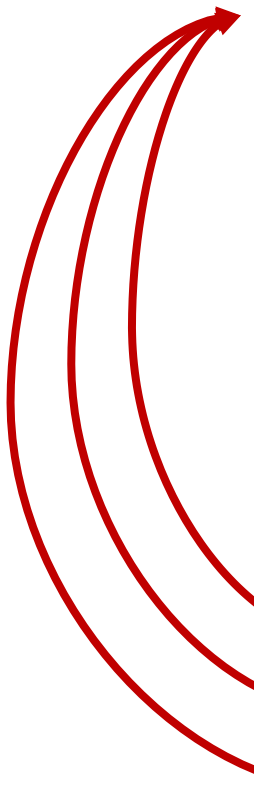
int a[5];

When sending the array (a), we send its **address**

When sending the element of array (a[2]), we send its **value**

Functions can be nested

```
#include<stdio.h>

int max(int x, int y);
int max_4(int a, int b, int c, int d);

main()
{
   int a = 20, b = 10, c = 4, d = 1;
   int z = max_4(a, b, c, d);
}
```



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

int max_4(int a, int b, int c, int d)
{
     int z;
     z = max(a, b);
     z = max(z, c);
     z = max(z, d);
     return z;
}
```

max_4() calls max() 3 times!!!

Functions can be nested

```
#include<stdio.h>

int add(int x, int y);
int sum(int a[]);

main()
{
   int a[] = {1,3,5,7,2,9,-3,2};
   int z = sum(a, sizeof(a)/sizeof(a[0]));
}
```

```
int add(int x, int y)
{
    return x + y;
}

int sum(int a[], int len)
{
     int z = 0;
     for(int i = 0; i < len; i++)
     {
        z = add(z, a[i]);
     }
     return z;
}
```

sum() repeatedly calls add() in a loop!!!

Functions can be mutually nested

```c
#include <stdio.h>

void sub(int a, int b);
void sum(int a, int b);

main()
{
    int a = 5, b = 10;

    sum(a, b);
}
```

```c
void sum(int a, int b)
{
    printf("a = %d\n", a);
    a = a + b;
    sub(a, b);
}
```

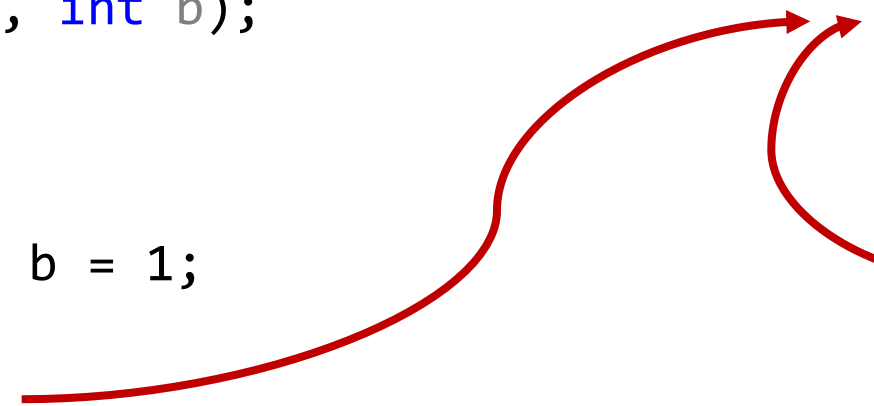
```c
void sub(int a, int b)
{
    a = a - b;
    sum(a, b);
}
```

Mutually call each other leads to a dead loop!!!

Function can be self-called

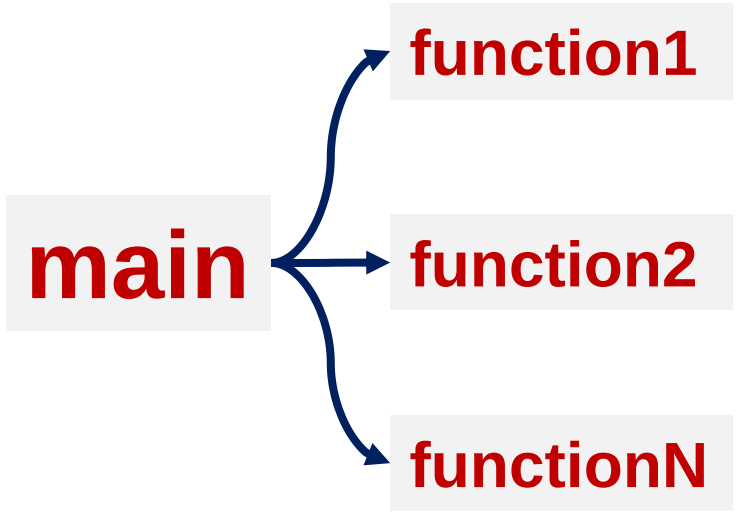
```c
#include <stdio.h>

void sum(int a, int b);

main()
{
    int a = 5, b = 1;

    sum(a, b);
}
```

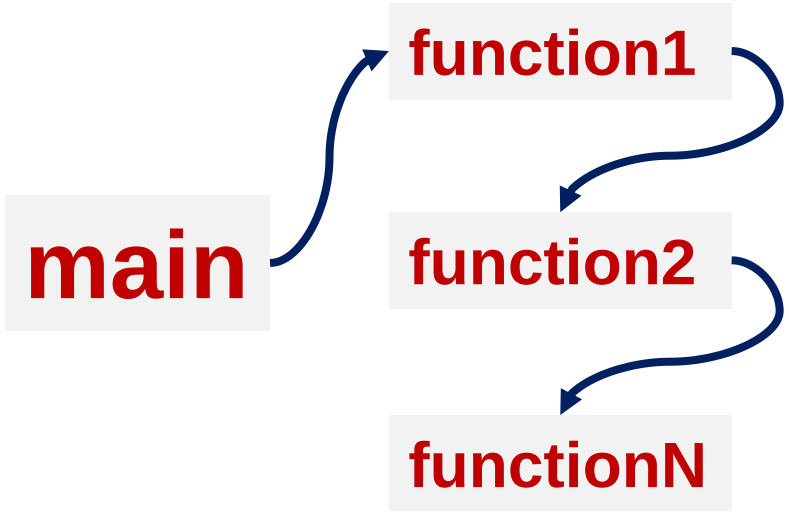
```c
void sum(int a, int b)
{
    printf("a = %d\n", a);
    a = a + b;
    sum(a, b);
}
```

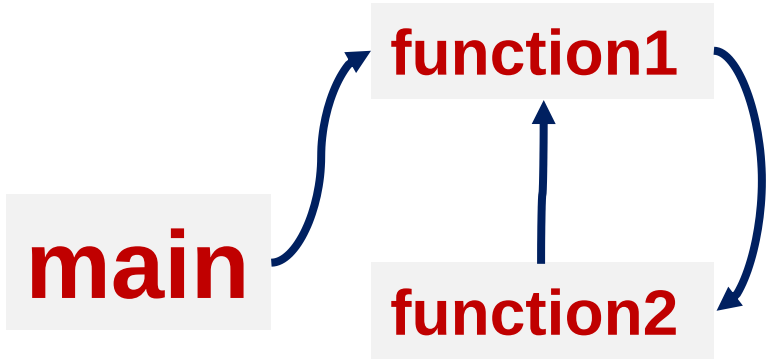


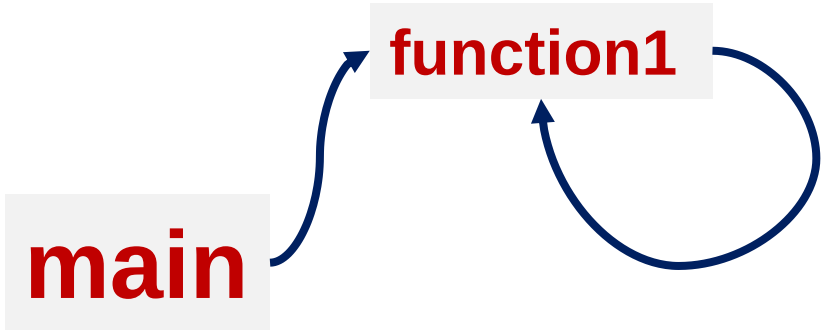
This is recursion!!!

Functions


main
function1
function2
functionN


main
function1
function2
functionN


main
function1
function2


main
function1

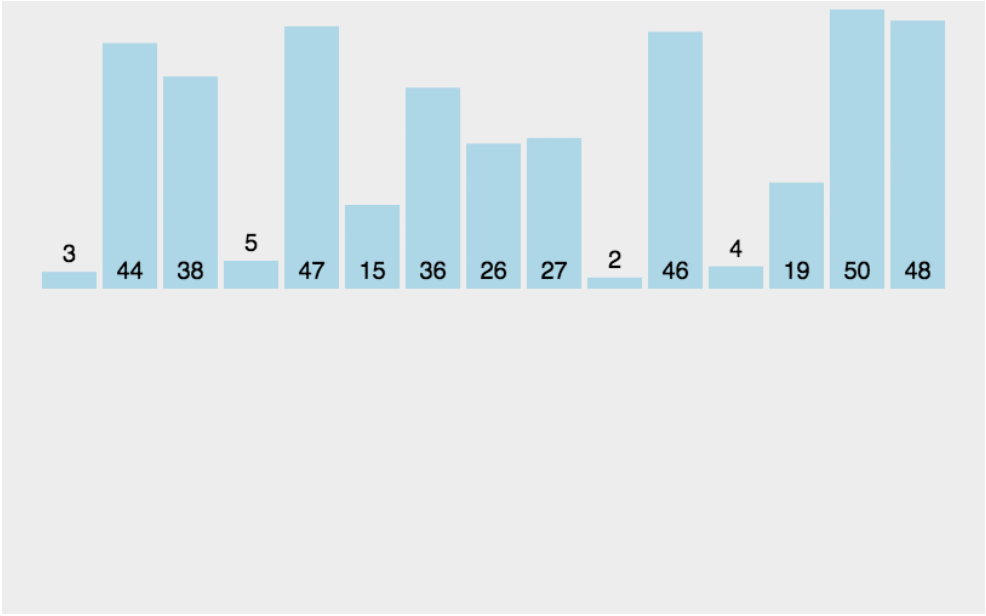
Case study: insertion sort

```c
void insertion_sort(int arr[], int len) {
    int i, j, key;
    for (i = 1; i < len; i++) {
        key = arr[i];
        j = i - 1;
        while ((j >= 0) && (arr[j] > key)) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

int main() {
    int arr[] = { 3,44,38,5,47,15,36,26,27,2,46,4,19,50,48};

    int len = (int)sizeof(arr) / sizeof(*arr);

    insertion_sort(arr,len);

    for (int k = 0; k < len; k++)
        printf("%d ", arr[k]);
    return 0;
}
```

Define function

Call function

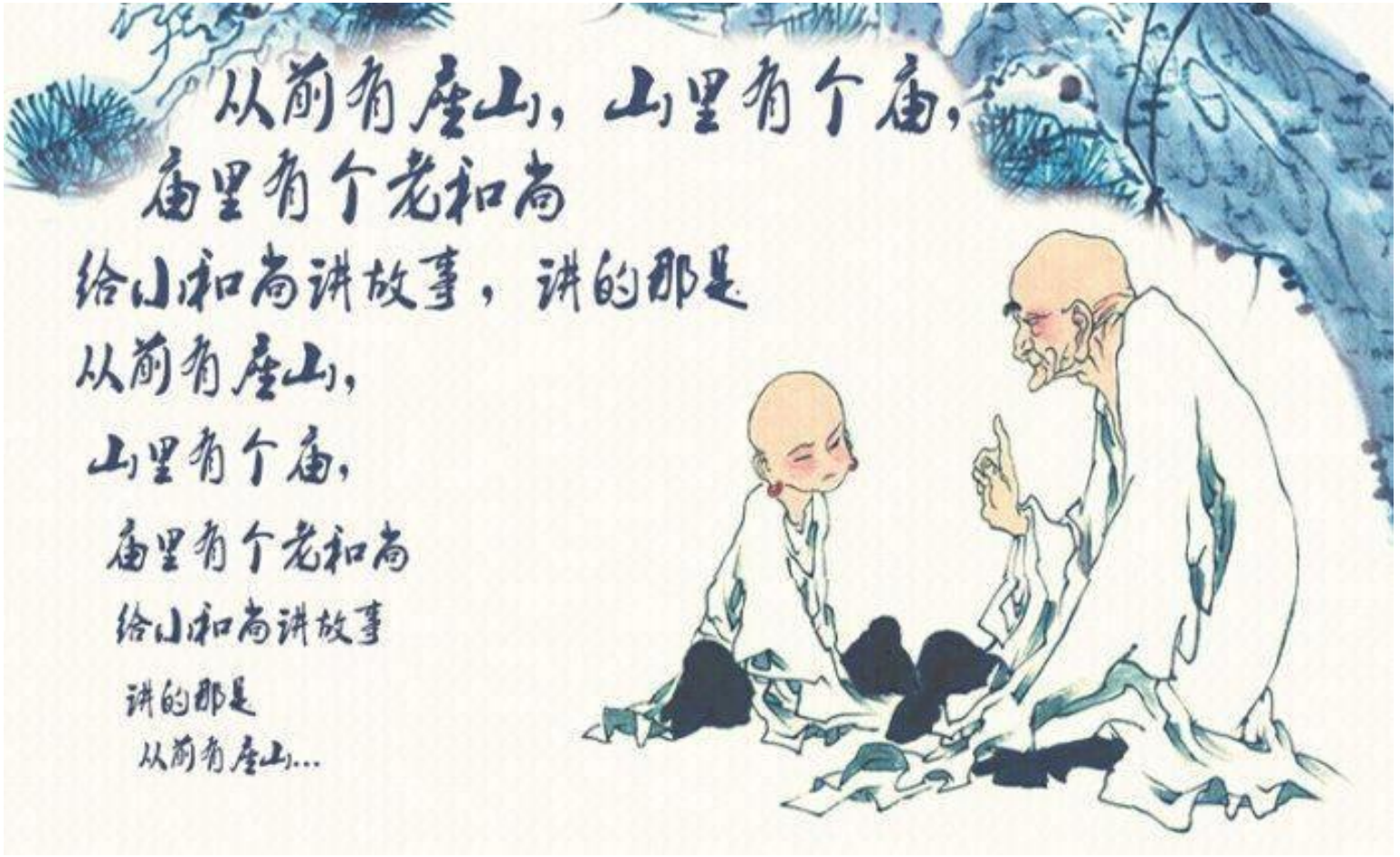
Case: create insertion sort algorithm!

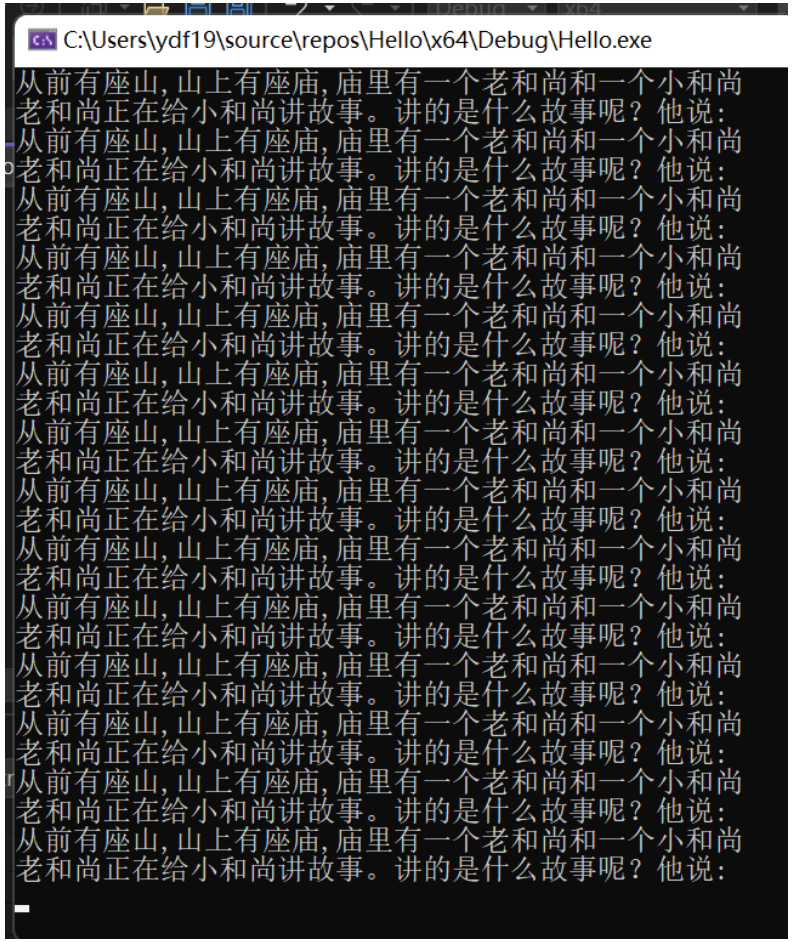



```
Microsoft Visual Studio Debug Console
2 3 4 5 15 19 26 27 36 38 44 46 47 48 50
C:\Users\wenji\Desktop\WorkStation\work\teaching\int
(process 44008) exited with code 0.
To automatically close the console when debugging st
le when debugging stops.
Press any key to close this window . . .
```

Case study: 从前有座山…

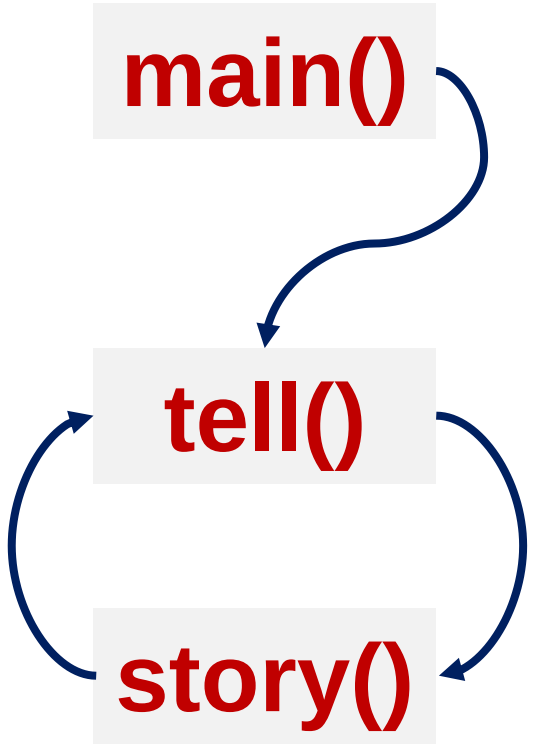
Case: repeat the same story again and again!





Case study: 从前有座山…

```c
#include <stdio.h>

void tell();
void story();

main()
{
    tell();
}

void story()
{
    printf("老和尚正在给小和尚讲故事。讲的是什么故事呢？他说:\n");
    tell();
}

void tell()
{
    printf("从前有座山,山上有座庙,庙里有一个老和尚和一个小和尚\n");
    story();
}
```



Case study: how many subjects got fever

```
int fever(float* tems, int size);

main()
{
    float temperature[5] = { 34.2,37.8,36.6,36.8,37 };
    int num_fever = fever(temperature, 5);
    printf("有 %d 个人发烧了",num_fever);
}

int fever(float* tems, int size)
{
    int num_fever = 0;
    for (int i = 0; i < size; i++)
    {
        if (tems[i] > 37.3)
            num_fever++;
    }
    return num_fever;
}
```

Case: find number of subjects that got the fever!

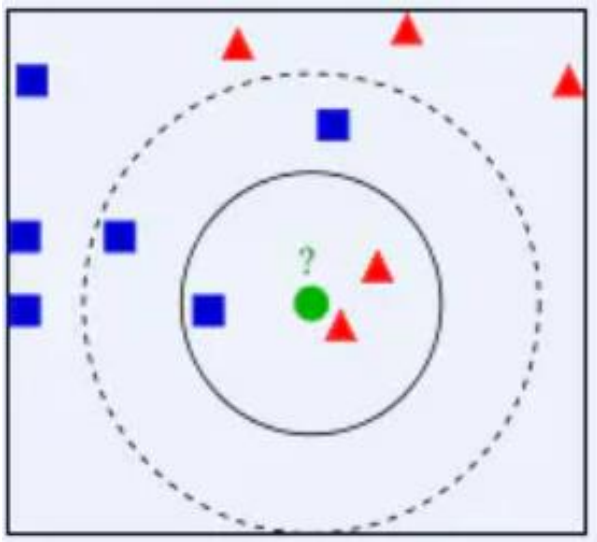
Microsoft Visual Stu

```
有 1 个人发烧了
C:\Users\ydf19\sou
```

Case study: find 3 nearest points

```
float dist(int x1, int y1, int x2, int y2);

main()
{
    int pts[5][2] = {{2,54},{45,67},{25,5},{23,62},{86,34}};
    int center[2] = { 25,25 };
    float D[5] = {};
    for (int i = 0; i < 5; i++)
        D[i] = dist(center[0],center[1],pts[i][0],pts[i][1]);

    insertion_sort(D, 5);

    for (int i = 0; i < 3; i++)
        printf("距离为 %f\n", distances[i]);
}

float distance(int x1, int y1, int x2, int y2)
{
    float distance = sqrt(pow(x2-x1, 2) + pow(y2-y1, 2));
    return distance;
}
```

Case: given the center location, find the nearest 3 points.




```
距离为 20.000000
距离为 37.013512
距离为 37.054016
```

Content

1. Declare/define/call a function
2. **Variable scope**
3. Recursion

Variable scope

Scope is a region of the program where defined variables are valid and beyond that variables cannot be accessed.

Global variable
（全局变量）

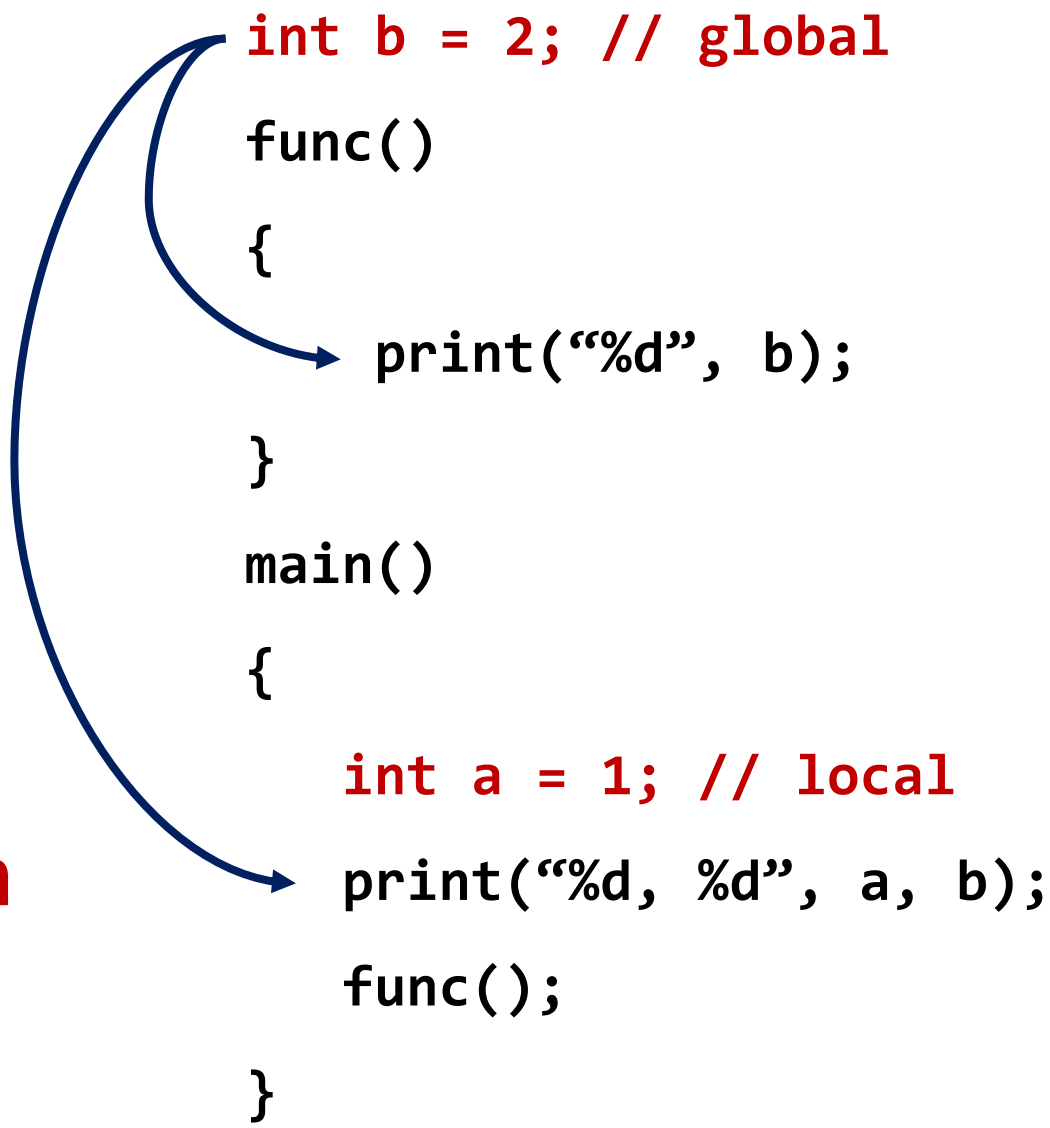
```
int b; //outside main

main()
{
    int a; //inside main
}
```

Local variable
（局部变量）

Variable scope

Global variable can be accessed everywhere

Local variable can only be accessed inside the function where it was defined



```
int b = 2; // global
func()
{
    print("%d", b);
}
main()
{
    int a = 1; // local
    print("%d, %d", a, b);
    func();
}
```

Variable scope

Global variable can be changed everywhere and keep the changes

```
int b = 2; // global
func()
{                          ②
    print(“%d”, b); //print 10
    b = 20;
}
main()
{                          ①
   print(“%d”, b); //print 2
   b = 10;
   func();                 ③
   print(“%d”, b); //print 20
}
```

Variable scope

- **Global and local variables can share the same name**
- **Local variable has the priority!!!**

```
int b = 2; // global
func()
{                          ②
    print("%d", b); //print 2
    b = 20;
}
main()
{
   int b = 5; // local   ①
   print("%d", b); //print 5
   b = 10;
   func();                 ③
   print("%d", b); //print 10
}
```

Variable scope

```
float PI = 3.14; // global
float func(float a)
{
    return a * PI;
}
main()
{
   float a = 5; // local
   float b = func(a);
   print(“%d”, b);
}
```

- ✓ **Do not use global variables unless**
  - It is a **constant** that can be used everywhere (consensus)
  - Its value needs to be **shared and changed** in multiple blocks or threads (e.g. bank account)
  - Limited **memory** resources (embedded system)
- ✓ **Use local variables as much as possible!**

Storage classes for variable

```
identifier int a = 5;
```

↓

自动 `auto int a = 5;`

→ 静态 `static int a = 5;`

外部 `extern int a = 5;`

寄存器 `register int a = 5;`

Auto variable

Memory for variable is automatically created when the function is invoked and destroyed when a block exits. **By default, local variables have automatic storage duration.**

```
#include <stdio.h>
main(){
    auto int i = 10;
    float j = 2.8;
}

void myFunction(){
    int a;
    auto int b;
}
```

Both i and j are auto variables

Both a and b are auto variables

Static variable

For static variables, memory is allocated only once and storage duration remains until the program terminates. **By default, global variables have static storage duration.**

```
#include <stdio.h>
int x = 1;
void increment()
{
   printf("%d\n", x);
   x = x + 1;
}
main()
{
  increment();  1
  increment();  2
}
```

```
#include <stdio.h>
void increment()
{
   int x = 1;
   printf("%d\n", x);
   x = x + 1;
}
main()
{
  increment();  1
  increment();  1
}
```

```
#include <stdio.h>
void increment()
{
   static int x = 1;
   printf("%d\n", x);
   x = x + 1;
}
main()
{
  increment();  1
  increment();  2
}
```

Extern variable

Extern can only be used to define global variables. An extern variable can be assessed across different C files.

```
/***> File Name: extern_test.c ***/
#include <stdio.h>
int ex_num = 20;
int num = 30;
char str[81] = "abcdefg";
```

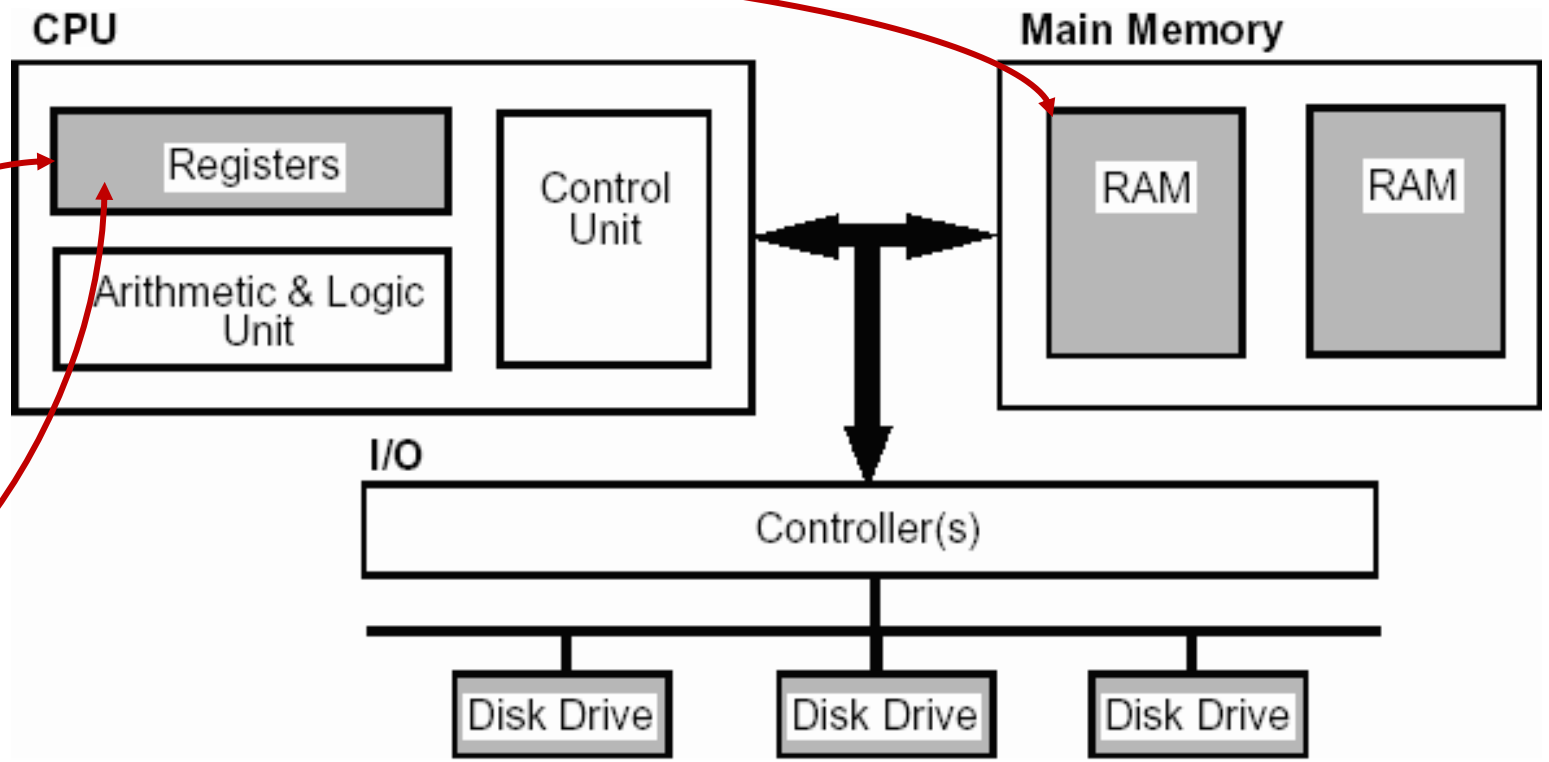
```
num = 30
ex_num = 20
str = abcdefg
c = 10
```

```
/***> File Name: main_test.c***/
#include <stdio.h>
extern int num;
extern int ex_num;;
extern char str[81];
int c = 10;
main(){
    printf("num = %d\n", num);
    printf("ex_num = %d\n", ex_num);
    printf("str = %s\n", str);
    printf("c = %d\n", c);
}
```

Register variable

Register is used to define local variables that are stored in a register (faster) instead of RAM. **By default, local variables have register storage.**

```
#include <stdio.h>

int b = 1;

myFunction(){
   int c = 1;
   register int d = 2;
}

main(){
   int a = 10;
}
```

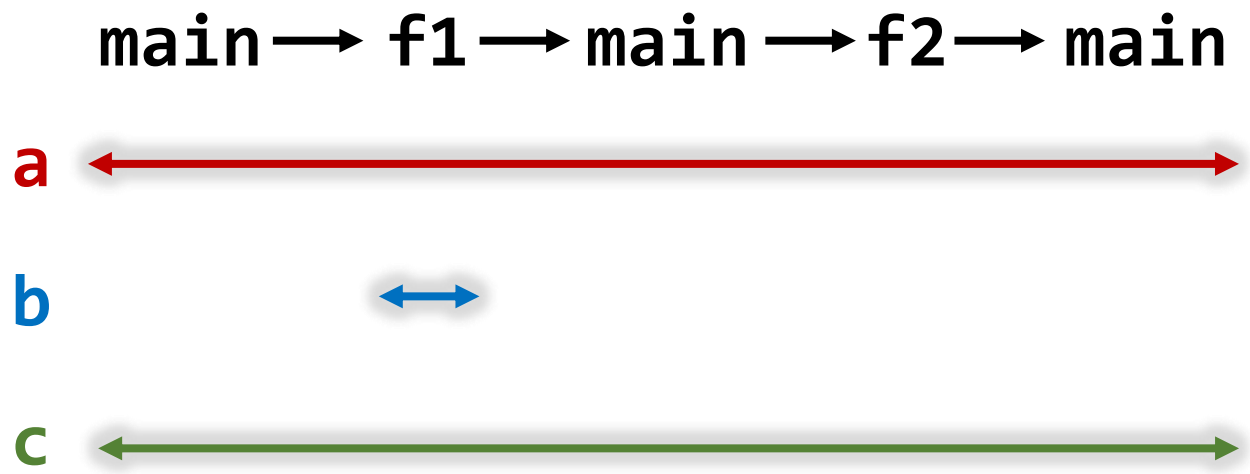


Variable scope

Scope in space

```
int a;
f1();
f2();
main()
{
   f1();
   f2();
}
f1(){int b;}
f2(){static int c;}
```

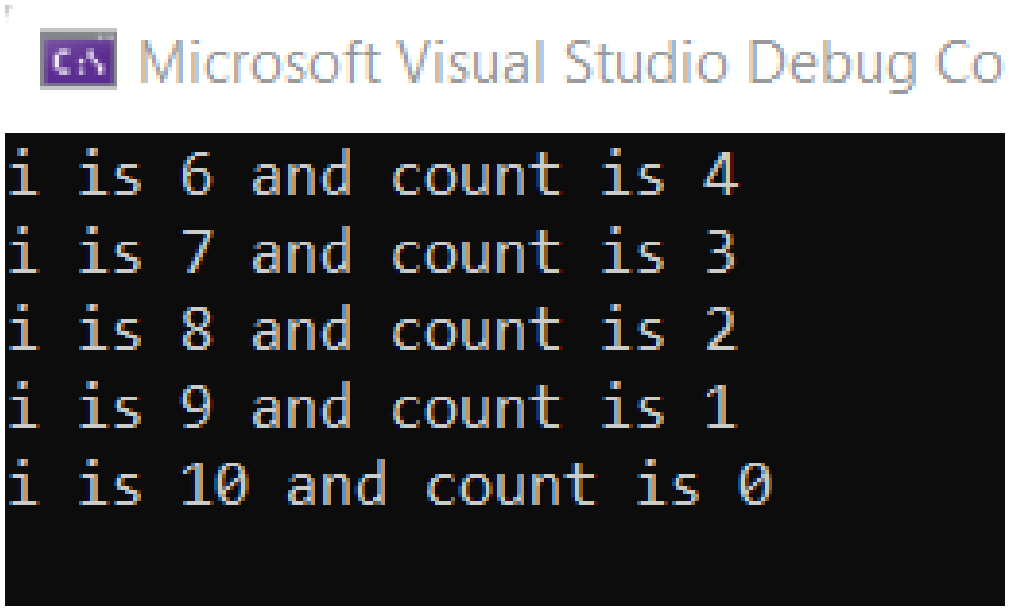
Scope in time



Case study: static variable

```c
#include <stdio.h>

void func(void);

static int count = 5;
main() {

    while (count--) {
        func();
    }

        return 0;
}

void func(void) {
    static int i = 5;
    i++;
    printf("i is %d and count is %d\n", i, count);
}
```

Case: create two static counters (increment and decrement).



Content

1. Declare/define/call a function
2. Variable scope
3. **Recursion**

Recursion in life

Recursion in life

Recursion in life

Recursion in life

Recursion

Recursion is to repeat the same procedure again and again

```
void recurse()
{
    recurse();
}
main()
{
    recurse();
}
```

Function call

Recursion

Recursively subtract

```
void recurse(int n)
{
    recurse(n-1);
}
main()
{
    int n = 100;
    recurse(n);
}
```

100

99, 98, 97,...,0,-1,...

Recursively add

```
void recurse(int n)
{
    recurse(n+1);
}
main()
{
    int n = 0;
    recurse(n);
}
```

0

1, 2, 3, ...,100, 101,...

Recursion

```
void recurse(int n)
{
    if (n == 0) return;
    recurse(n-1);
}
main()
{
     int n = 100;
     recurse(n);
}
```

Which one can leave?

```
void recurse(int n)
{
     recurse(n-1);
     if (n == 0) return;
}
main()
{
     int n = 100;
     recurse(n);
}
```

Recursion

```
void recurse(int n)
{
    if (n == 0) return;
    recurse(n-1);
}
main()
{
     int n = 100;
     recurse(n);
}
```

Which one is better?

```
main()
{
     int n = 100
     for(; n > 0; n--)
     {
     }
}
```

Case study: factorial calculator

```c
#include <stdio.h>

double factorial(int i)
{
    if (i <= 1)
    {
        return 1;
    }
    return i * factorial(i - 1);
}

main()
{
    int input;
    scanf("%d", &input);
    printf("The Factorial of %d is %f\n", input,
factorial(input));
}
```

Case: use recursion to design a factorial calculator.

$$n! = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n$$

$$n! = n \times (n-1)!$$

```
3
The Factorial of 3 is 6.000000
```

```
5
The Factorial of 5 is 120.000000
```

Case study: Fibonacci series

```c
#include <stdio.h>

int fib(int input) {
    if (input <= 2) {
        return 1;//first two numbers are 1
    }
    return fib(input - 1) + fib(input - 2);
}

main() {
    int input = 0;
    scanf("%d", &input);
    fib(input);
    printf("The No.%d Fibonacci number is :%d",
input, fib(input));
}
```

Case: use recursion to implement a Fibonacci series.

$F(1)=1, F(2)=1,$

$F(n)=F(n-1)+F(n-2)\ (n\geq 3,\ n\in N^*)$

1、1、2、3、5、8、13、21、34、…

```
3
The No.3 Fibonacci number is :2
```

```
6
The No.6 Fibonacci number is :8
```

Case study: Hanoi tower

一个源于印度的古老传说：大梵天创造世界的时候做了三根石柱子，在一根柱子按照大小顺序摞着**64片黄金圆盘**。大梵天命令婆罗门把圆盘从下面开始按大小顺序重新摆放在另一根柱子上。并且规定，在小圆盘上不能放大圆盘，在三根柱子之间一次只能移动一个圆盘。

大梵天说：**当移动完所有黄金圆盘，将海枯石烂，天荒地老。**

Case study: Hanoi tower

3 disks:

4 disks:

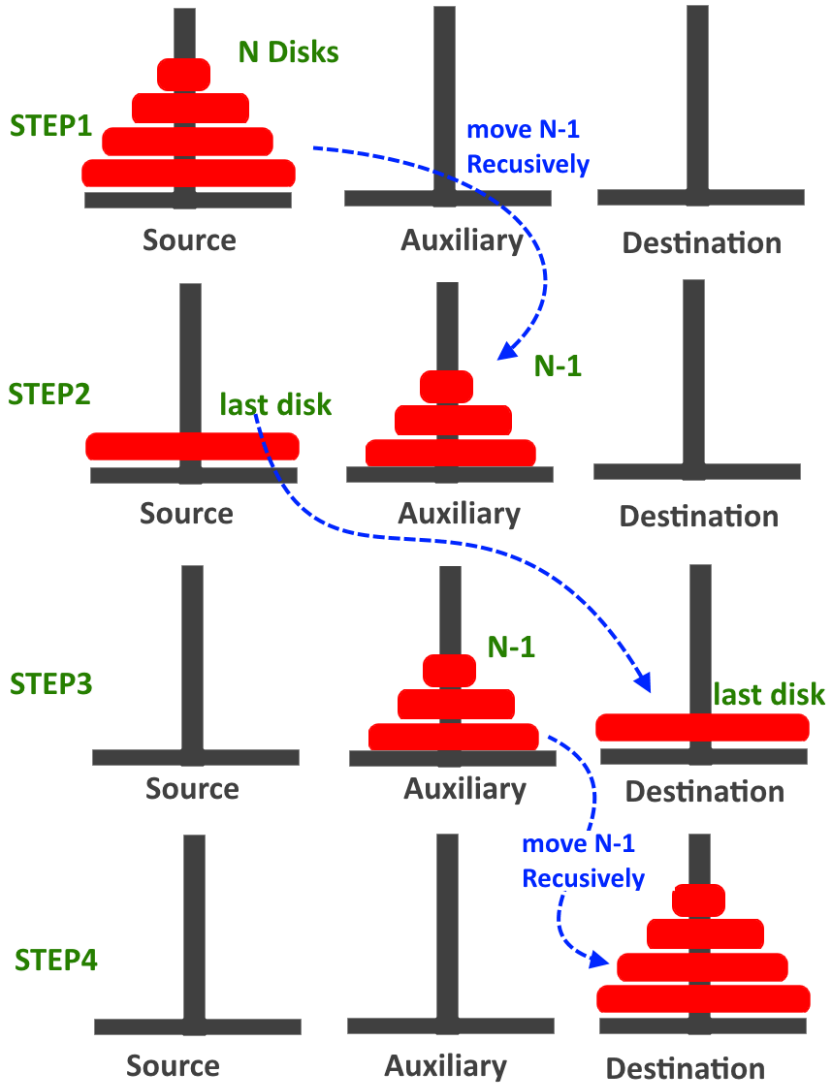
Hanoi tower rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
3. No disk can be placed on top of a disk that is smaller than it.

Assume n disks, move time is f(n), f(1)=1,f(2)=3,f(3)=7 f(k+1)=2*f(k)+1, total moves is f(n)=2^n-1, n=64 means 2^{64} - 1, 1 sec 1 disk, **5845.42亿年** 以上，而地球存在至今不过45亿年!!!

Case study: Hanoi tower

```c
#include<stdio.h>
void move(char A, char C, int n){
    printf("Move disc %d from %c to --->%c\n", n, A, C);
}
void HanoiTower(char A, char B, char C, int n){
    if (n == 1){
        move(A, C, n);
    }
    else{
        HanoiTower(A, C, B, n - 1); //Move n-1 discs
    from peg A to peg B using extra peg C
        move(A, C, n); //Move the No. n peg from A to C
        HanoiTower(B, A, C, n - 1); //Move n-1 discs
    from peg B to peg C using extra peg A
    }
}
main(){
    int n = 0;
    printf("Input the number of pegs on disc A: ");
    scanf("%d", &n);
    HanoiTower('A', 'B', 'C', n);
}
```



Case study: Hanoi tower

```c
#include<stdio.h>
void move(char A, char C, int n){
    printf("Move disc %d from %c to --->%c\n", n, A, C);
}
void HanoiTower(char A, char B, char C, int n){
    if (n == 1){
        move(A, C, n);
    }
    else{
        HanoiTower(A, C, B, n - 1); //Move n-1 discs
    from peg A to peg B using extra peg C
        move(A, C, n); //Move the No. n peg from A to C
        HanoiTower(B, A, C, n - 1); //Move n-1 discs
    from peg B to peg C using extra peg A
    }
}
main(){
    int n = 0;
    printf("Input the number of pegs on disc A: ");
    scanf("%d", &n);
    HanoiTower('A', 'B', 'C', n);
}
```

```
Input the number of pegs on disc A: 1
Move disc 1 from A to --->C
```

```
Input the number of pegs on disc A: 2
Move disc 1 from A to --->B
Move disc 2 from A to --->C
Move disc 1 from B to --->C
```

```
Input the number of pegs on disc A: 3
Move disc 1 from A to --->C
Move disc 2 from A to --->B
Move disc 1 from C to --->B
Move disc 3 from A to --->C
Move disc 1 from B to --->A
Move disc 2 from B to --->C
Move disc 1 from A to --->C
```

Summary

1. Declare/define/call a function
2. Variable scope
3. Recursion

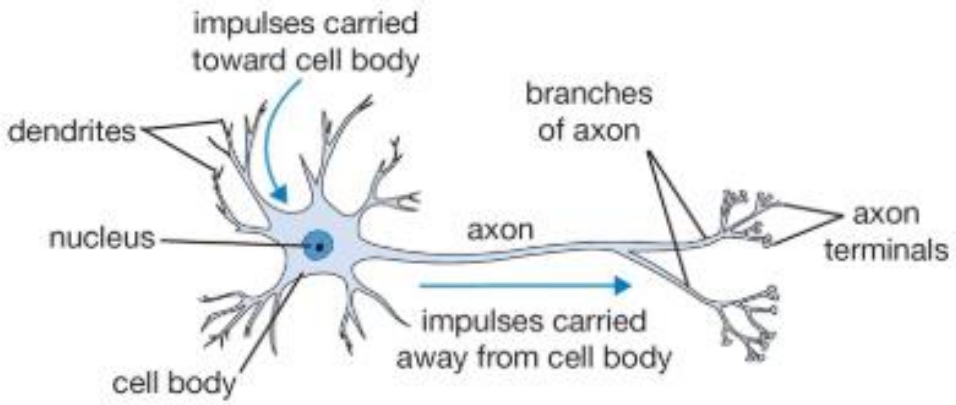
Summary

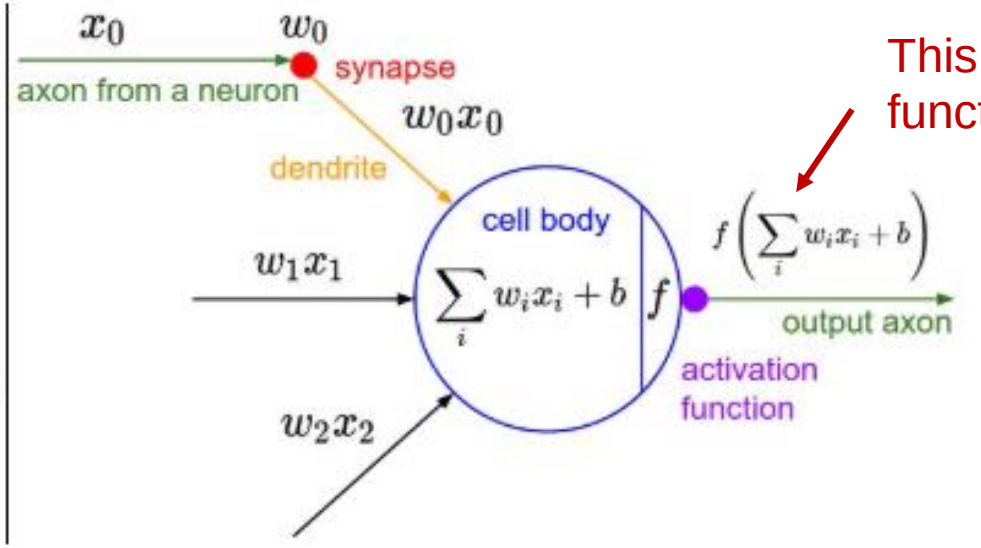
- We can create our own functions, the procedure includes 3 steps: **function declaration, definition, calling**.
- Function declaration and definition can be merged, but needs to be in front of the place where it is called (e.g. main)
- Variable has its scope, both in space and time. **Global variable (outside function)** is visible everywhere, **local variable (inside function)** is only visible in the function block.
- **Recursion** can be implemented by calling a function itself repeatedly.
- Time to write you own functions in C!!!

Assignment

1. Neuron is the core of artificial neural network. The activation function is a gating function that maps linear input of the neuron to a non-linear output. There are four commonly used activation functions. Write these functions in C and call them in main.

a) Use “scanf” to enter a number as argument of the function
b) Test input : -0.5, 9
c) Float input and float output

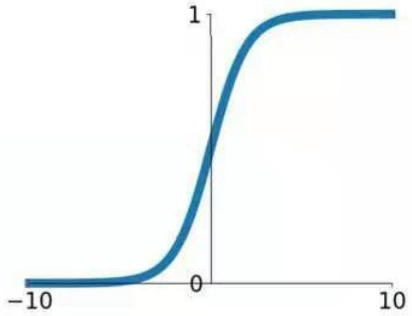




Assignment

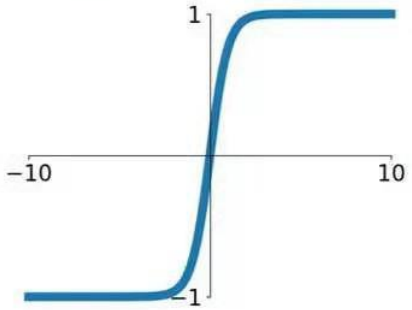
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



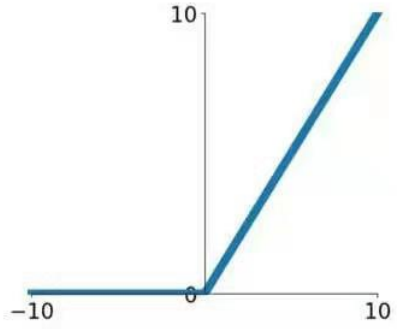
tanh

$$\tanh(x)$$



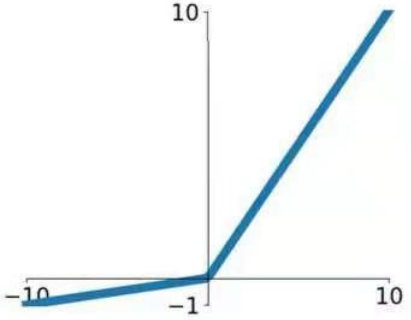
ReLU

$$\max(0, x)$$



Leaky ReLU

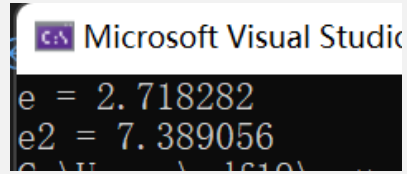
$$\max(0.1x, x)$$



$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

How to calculate e^{x} in C ?

```c
#include<stdio.h>
#include<math.h>

int main(void)
{
    float e = exp(1);
    float e2 = exp(2);
    printf("e = %f\ne2 = %f",e,e2);
    return 0;
}
```

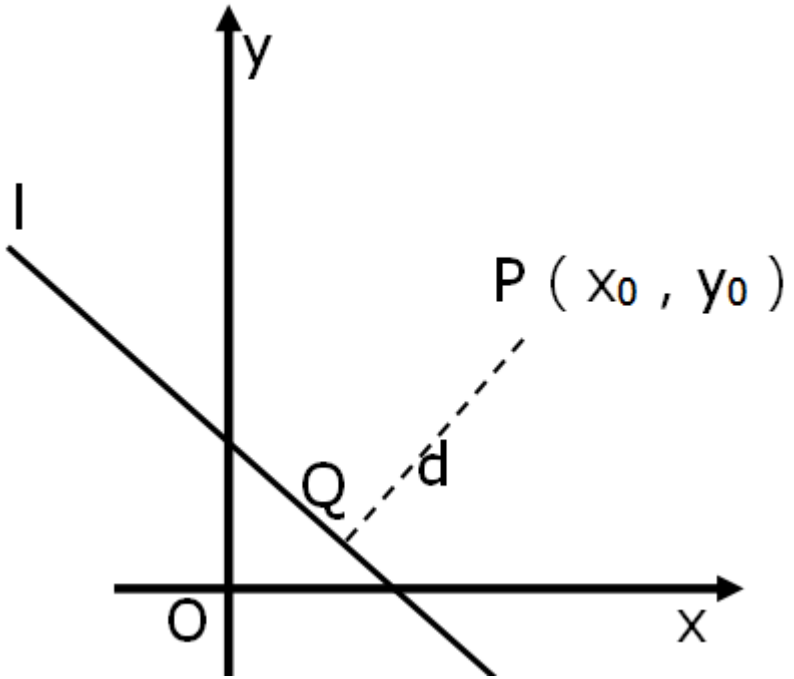


Assignment

2. Write a function to calculate the distance between a point (1, 5) and a straight line 2x + 6y + 8 = 0, return the distance and call this function in main.

a) The distance is a float number

b) Use scanf to enter the point (1, 5)



Ax + By + C = 0

$$d = \left| \frac{Ax_0 + By_0 + C}{\sqrt{A^2 + B^2}} \right|$$

Assignment

3. Enter an integer and print each decimal bit of the integer using recursion(e.g. enter 12345 and print 1,2,3,4,5)

a) Use “scanf” to enter the number

b) Test input: 12345

```
#include<stdio.h>

void cal_bits(int)
{
    //计算出一位
    printf(“%d ”,bit);//打印一位
    cal_bits(int)//递归
}
```

Assignment

4. Write a function to implement the dot-product between two matrices (A · B) and call the function in main

a) Use two int [] (or int*) as the arguments of the function and return int [] (or int*)

b) A and B are 2D matrices

b) Int input and int output

c) Test input: $A = \begin{matrix} 1 & 2 \\ 3 & 4 \end{matrix}$ $B = \begin{matrix} 5 & 6 \\ 7 & 8 \end{matrix}$

Refer to slide 49

$$\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix} \cdot \begin{matrix} 10 & 11 & 12 \\ 13 & 14 & 15 \\ 16 & 17 & 18 \end{matrix} = \begin{matrix} 1*10 & 2*11 & 3*12 \\ 4*13 & 5*14 & 6*15 \\ 7*16 & 8*17 & 9*18 \end{matrix}$$

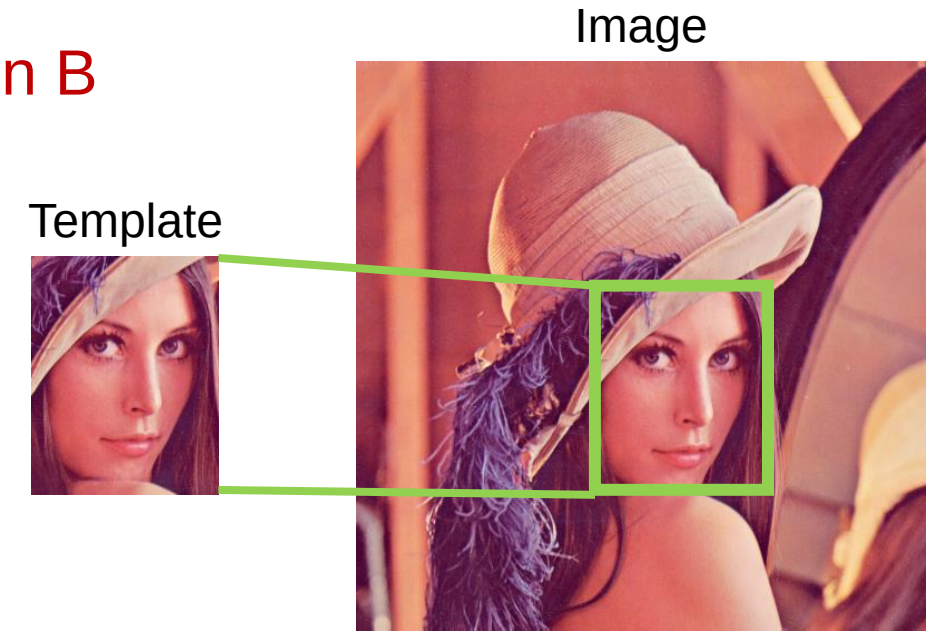
Assignment

5. Target searching is commonly used in Computer Vision (e.g. face detection). Target searching can be implemented by template matching (or cross-correlation): use a pre-defined 2D matrix as the template and find its matching location in a larger 2D matrix. In this process, template is used to slide over the larger 2D matrix and calculate the correlations per slide. Correlation is defined as the sum of multiplied elements of two matrices.

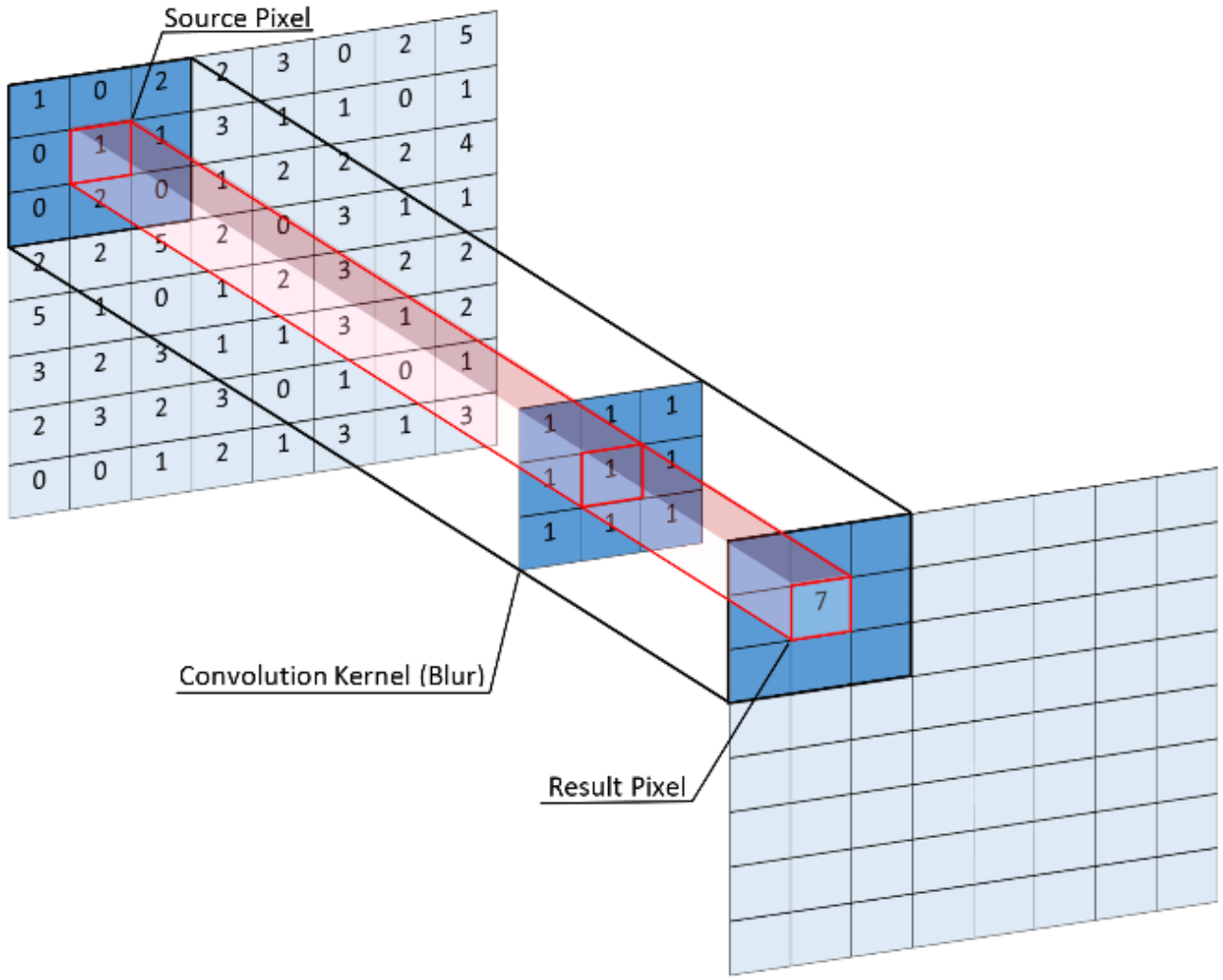
a) The template is A, the image is B
b) Output the best matching location (row, column) in B

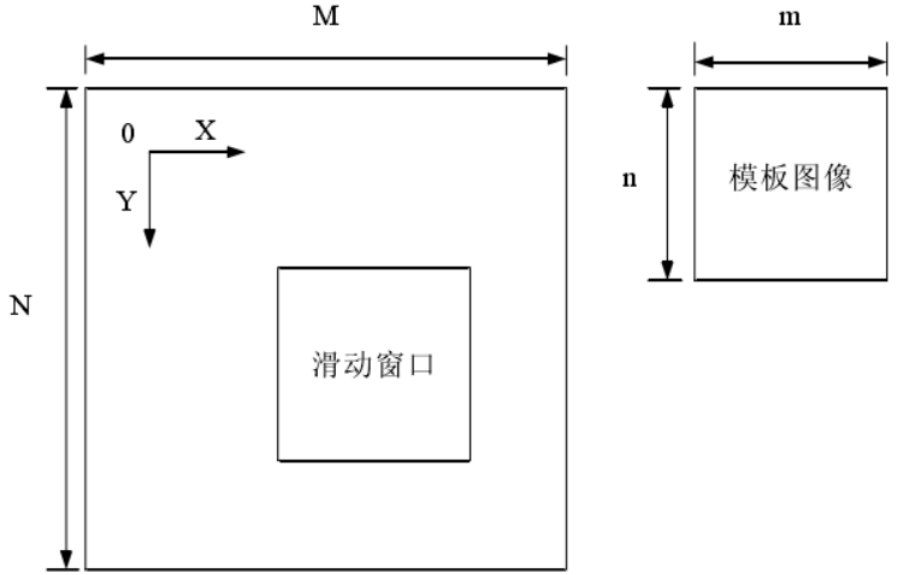
$$A = \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix}$$

$$B = \begin{matrix} 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \end{matrix}$$



Assignment


Source Pixel
Convolution Kernel (Blur)
Result Pixel


M
m
0
X
Y
N
n
模板图像
滑动窗口