



C程序设计基础

Introduction to C programming Lecture 9: Functions

张振国 zhangzg@sustech.edu.cn

南方科技大学/理学院/地球与空间科学系

Review on L8 Array II

array

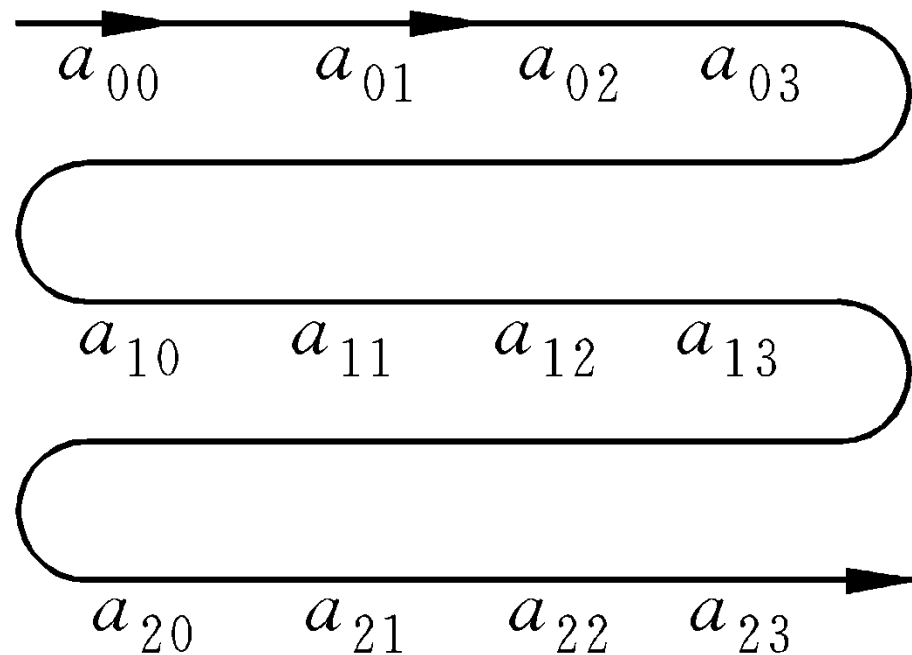
2-D and N-D array

String

Row-major or column-major order

2-D array

二维数组中的元素在内存中的排列顺序是：按行存放，即先顺序存放第一行的元素，再存放第二行的元素……



String

String is an array of characters.

```
char name[size] = { '\0', '\0', ..., '\0' };
```

```
char name[size] = {"..."};
```

```
char name[] = { "..."};
```

```
char name[] = "...";
```

String

```
char c[10] = {'S', 'U', 'S', 'T', 'e', 'c', 'h'}; // length is 10
```

```
char c[10] = {"SUSTech"};
```

```
char c[] = {"SUSTech"};
```

```
char c[] = "SUSTech"; // preferred
```

S	U	S	T	e	c	h	\0	\0	\0
c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]	c[8]	c[9]

IO - scanf

(1) 如果利用一个scanf函数输入多个字符串，则在输入时以空格分隔。

```
char str1 [5] , str2 [5] , str3 [5] ;  
scanf(" %s %s %s" , str1, str2, str3);
```

输入数据:

How are you?

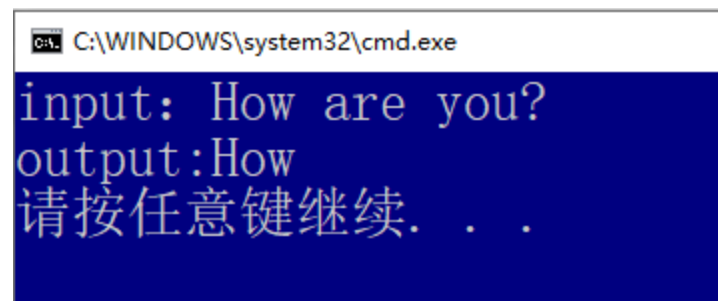
数组中未被赋值的元素的值自动置' \0' 。

H	o	w	\0	\0
a	r	e	\0	\0
y	o	u	?	\0

IO - scanf

(2) 如果利用一个scanf函数输入的字符串中若输入空格，其后都为'\0'

```
int main()
{
    char str[13];
    printf("input: ");
    scanf("%s", str);
    printf("output:%s", str);
}
```

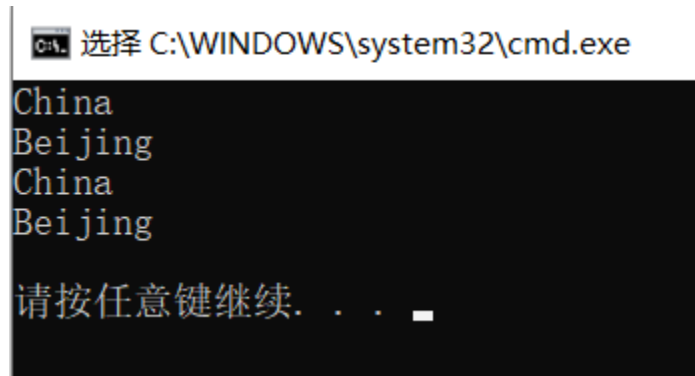


H	o	w	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0
---	---	---	----	----	----	----	----	----	----	----	----	----

IO - puts

其作用是将一个字符串 (以' \0' 结束的字符序列) 输出到终端。

```
#include <stdio.h>
int main()
{
    char str[] = {"China\nBeijing"};
    puts(str);
    puts(str);
    return 0;
}
```

A screenshot of a Windows command prompt window. The title bar reads "选择 C:\WINDOWS\system32\cmd.exe". The command prompt shows the output of the program: "China" on the first line, "Beijing" on the second line, "China" on the third line, and "Beijing" on the fourth line. Below the output, it says "请按任意键继续. . . " followed by a cursor.

在输出时，将字符串
结束标志' \0' 转换成' \n' ，
即输出完字符串后换行。

IO - gets

其作用是从终端输入一个字符串到字符数组，并且得到一个函数值。
该函数值是字符数组的起始地址。

```
#include <stdio.h>
int main()
{
    char str[10];
    gets(str);
    puts(str);
    return 0;
}
```

C:\WINDOWS\system32\cmd.exe

computer
computer

请按任意键继续

函数值为字符数组str的起始地址。一般利用gets函数的目的是向字符数组输入一个字符串，而不大关心其函数值。

注意：用puts和gets函数只能输入或输出一个字符串，不能写成puts(str1, str2) 或 gets(str1, str2)!!

String operations

```
#include <string.h>
```


C supports a wide range of functions that manipulate strings.

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 s1	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


strcpy(s1, s2)

```
char str1[12] = "Hello";  
char str2[12] = "World";  
char str3[12];
```

```
strcpy(str3, str1);  
printf("%s\n", str3); //Hello
```



```
strcpy(str3, str2);  
printf("str3 = %s\n", str3); //World
```



strcat(s1, s2)

```
char str1[12] = "Hello";  
char str2[12] = "World";  
char str1[12] = "123";
```

```
strcat(str1, str2);  
printf("str1 = %s\n", str1); //HelloWorld
```

```
strcat(str3, str2);  
printf("str3 = %s\n", str3); //123World
```

strlen(s1)

```
char str1[12] = "Hello";  
char str2[] = "World";  
char str3[12];
```

测试字符串长度的函数。函数的值为字符串中的实际长度(不包括' \0' 在内)。

```
printf("str1 = %d\n", strlen(str1)); //5
```

```
printf("str2 = %d\n", strlen(str2)); //5
```

```
printf("str3 = %d\n", strlen(str3)); //0
```

sizeof(s1)

```
char str1[12] = "Hello";  
char str2[] = "World";  
char str3[12];
```

```
printf("str1 = %d\n", sizeof(str1)); //12
```

```
printf("str2 = %d\n", sizeof(str2)); //6, end with '\0'
```

```
printf("str3 = %d\n", sizeof(str3)); //12
```

strcmp(s1, s2)

```
char str1[] = "ABCD";
```

```
char str2[] = "BCD";
```

```
char str3[] = "ABCE";
```

```
char str4[] = "1234";
```

str1 > str2 → 1

str1 < str2 → -1

str1 = str2 → 0

若出现不相同的字符，以第1对不相同的字符的比较结果为准

```
printf("cmp = %d\n", strcmp(str1, str2)); // -1
```

```
printf("cmp = %d\n", strcmp(str1, str3)); // -1
```

```
printf("cmp = %d\n", strcmp(str1, str1)); // 0
```

strcmp(s1, s2)

Application scene:

```
if (strcmp(s, "+") == 0) {  
    c = a + b;  
} else if (strcmp(s, "-") == 0) {  
    c = a - b;  
} else if (strcmp(s, "*") == 0) {  
    c = a * b;  
} else if (strcmp(s, "/") == 0) {  
    c = a / b;  
} else {  
    printf("ERROR");  
    return 0;  
}
```

It's useful when we have different parameters.

It is a very simple calculator.

run **L08_strcmp.c**

you can write code with much complexity

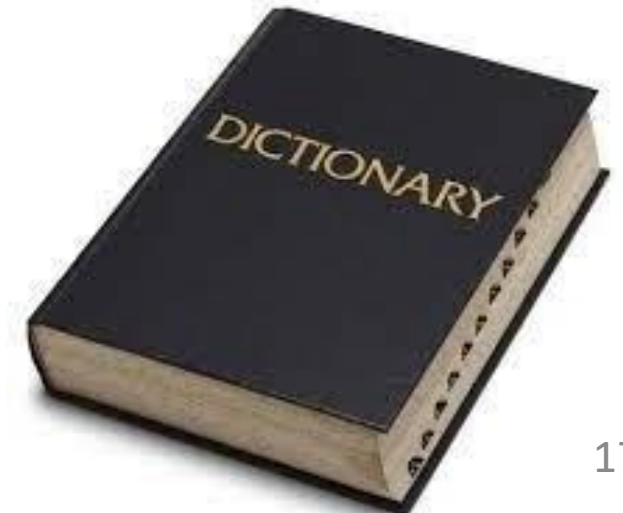


Case study: dictionary

Case: can we create a simple dictionary?

```
#include <stdio.h>
#include <string.h>
int main(void) {
    char EngWords[][8] = { "apple", "orange", "banana" };
    char ChineseWords[][8] = { "苹果", "橘子", "香蕉" };
    char text[128];
    while (gets(text)) {
        for (int i = 0; i < 3; i++) {
            if (strcmp(text, EngWords[i]) == 0) {
                printf("%s 中文为: %s\n", text, ChineseWords[i]);
                break; }
            else if (strcmp(text, ChineseWords[i]) == 0) {
                printf("%s 英文为 %s\n", text, EngWords[i]);
                break; }
        }
        if (strcmp(text, "exit") == 0) break;
    }
}
```

```
Z:\Courses\CS111\Code\L08_dict.exe
apple
apple 中文为: 苹果
香蕉
香蕉 英文为 banana
orange
orange
orange 中文为: 橘子
exit
-----
Process exited after 34.49 seconds with return value 0
```



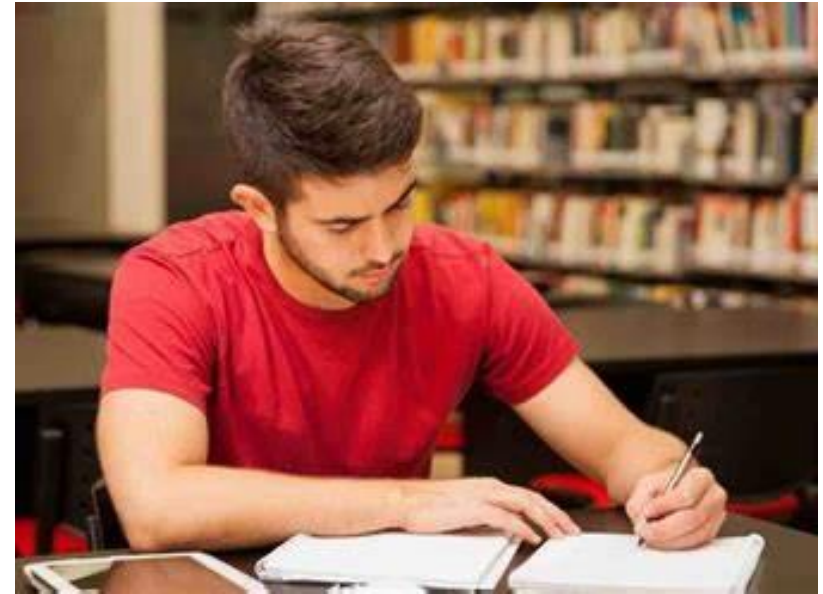
run **L08_dict.c**, **L08_dict2.c**

Case study: student's information

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char stu_name[][8] = { "张三", "李四", "王五" };
    char stu_id[][8] = { "111", "112", "113" };
    char text[128];
    while (gets(text)) {
        for (int i = 0; i < 3; i++) {
            if (strcmp(text, stu_name[i]) == 0) {
                printf("%s 学号为: %s\n", text, stu_id[i]);
                break; }
            else {printf(" % s 不在名单当中! \n", text);
                break; }
        }
    }
    return 0;
}
```

run **L08_studentIF.c**



Case study: encryption

Case: can we encrypt a message?

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char text[128] = { '\0' };
    char cryptograph[128] = { '\0' };
    printf("请输出要加密的明文: \n");
    gets(text);
    int count = strlen(text);
    for (int i = 0; i < count; i++) {
        cryptograph[i] = text[i] + i + 5;
    }
    cryptograph[count] = '\0';
    printf("加密后的密文是\n:%s", cryptograph);
}
```

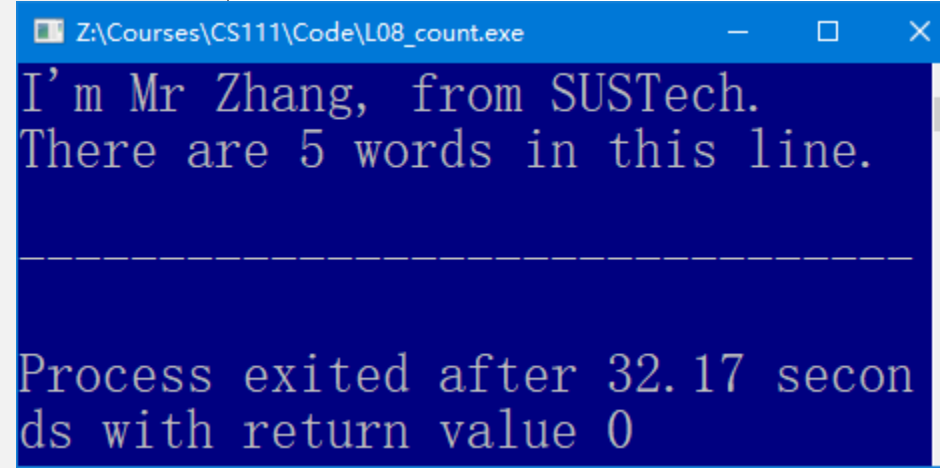
请输出要加密的明文:
Hello, Sustech!
加密后的密文是
:Mkstx6`亏俵sy24



Case study: count

Case: can we count how many words there are?

```
#include <stdio.h>
int main(void){
    char string[81];
    int i, num = 0, word = 0;
    char c;
    gets(string);
    for (i = 0; (c = string[i]) != '\0'; i++)
        if (c == ' ')
            word = 0;
        else if (word == 0)
        {
            word = 1;
            num++;
        }
    printf("There are %d words in this line.\n", num);
    return 0;
}
```



```
Z:\Courses\CS111\Code\L08_count.exe
I'm Mr Zhang, from SUSTech.
There are 5 words in this line.
-----
Process exited after 32.17 seconds with return value 0
```

check the
variable **word**
with
L08_count.c

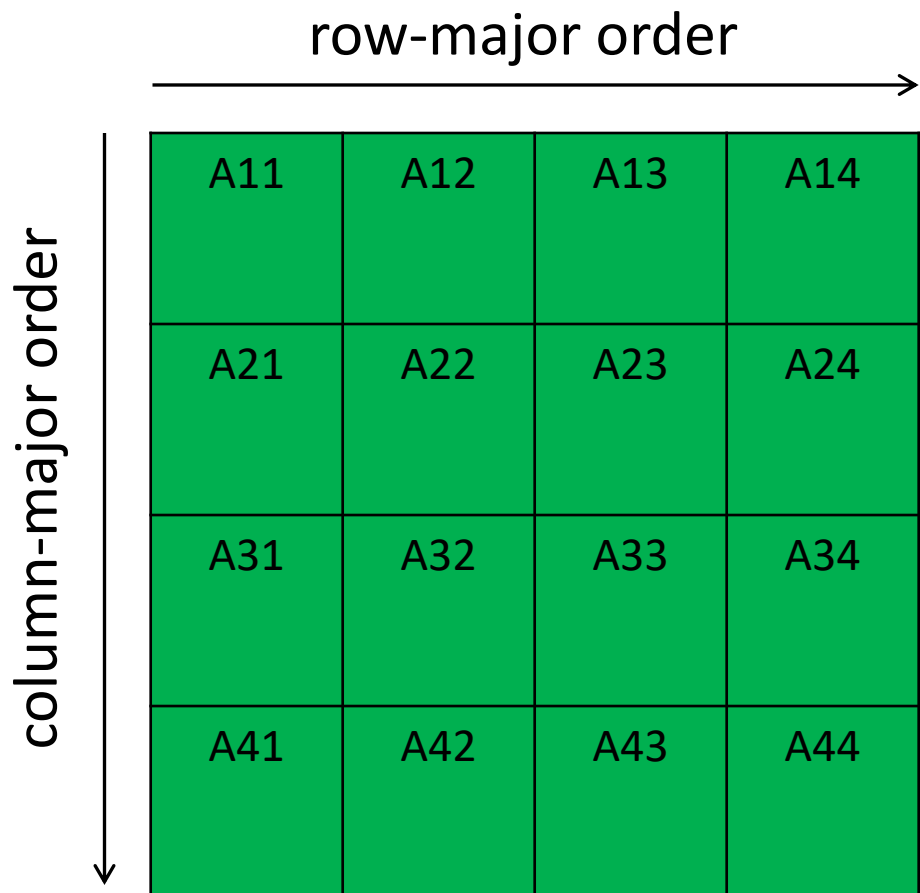
const

有时程序只需要从数组中读取数值，但是程序不向数组中写数据。在这种情况下，声明并初始化数组时，建议使用关键字**const**

```
const int days[MONTHS] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

程序会把每个元素当成常量来处理，在声明的时候需要对其初始化，因为初始化后，不能再对它赋值。

Row-major order or column-major order?



The diagram shows a 4x4 matrix with elements A11 through A44. A horizontal arrow above the matrix points to the right and is labeled 'row-major order'. A vertical arrow to the left of the matrix points downwards and is labeled 'column-major order'.

A11	A12	A13	A14
A21	A22	A23	A24
A31	A32	A33	A34
A41	A42	A43	A44

对于C语言来说，访问二维数组的顺序不同，时间消耗也是不同的。行优先遍历和列优先遍历进行对比，行优先更佳。

接下来通过C语言访问一个二维数组赋值操作来说对比优先与列优先的时间消耗差异，并给出相应的代码例子。

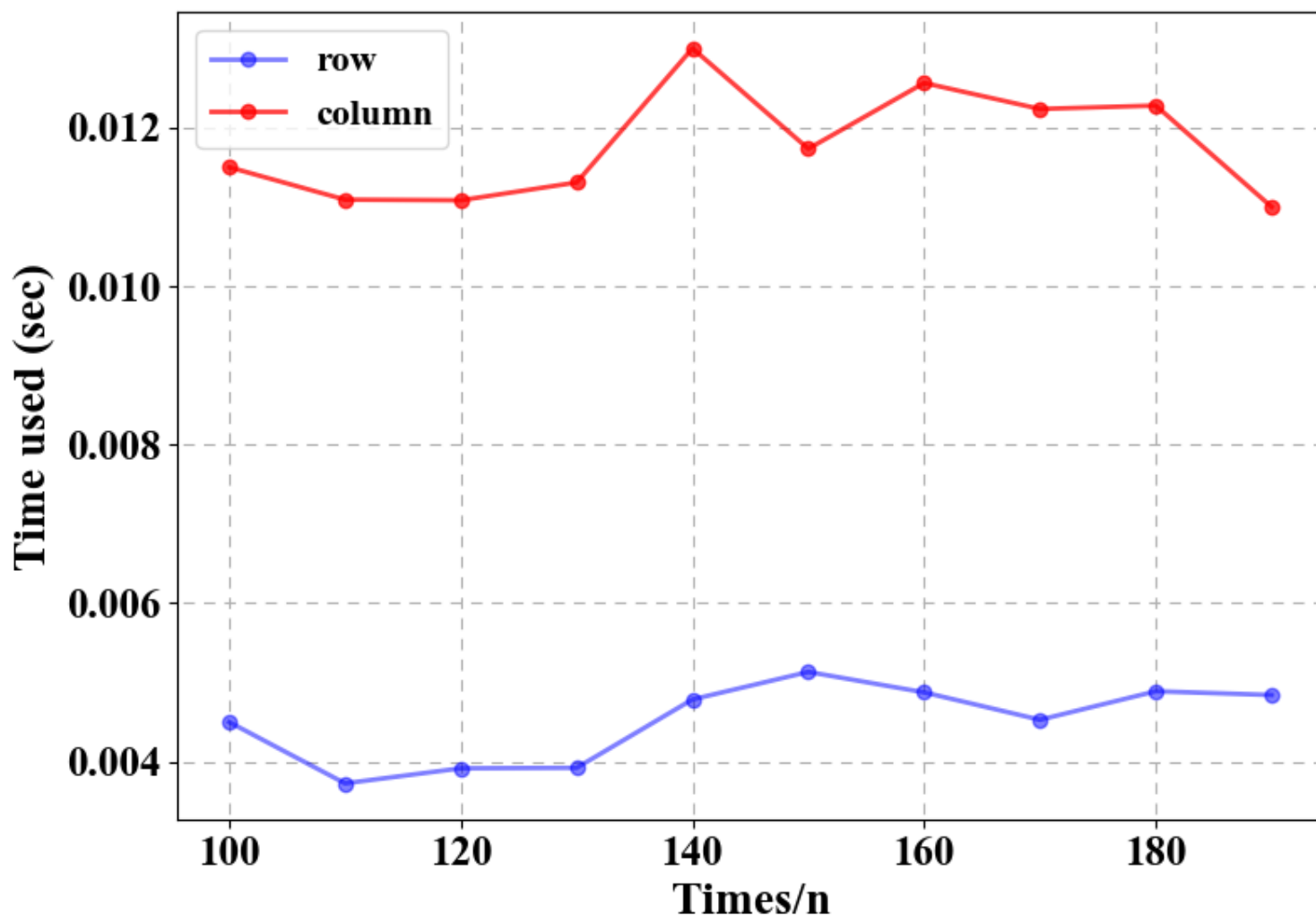
2-D array

整型数组 $b[3][3]=\{ \{1,2,3\}, \{4,5,6\}, \{7,8,9\} \};$

地址	值	数组元素
3000H	1	$b[0][0]$
3002H	2	$b[0][1]$
3004H	3	$b[0][2]$
3006H	4	$b[1][0]$
3008H	5	$b[1][1]$
300AH	6	$b[1][2]$
300CH	7	$b[2][0]$
300EH	8	$b[2][1]$
3010H	9	$b[2][2]$

Row-major order or column-major order?

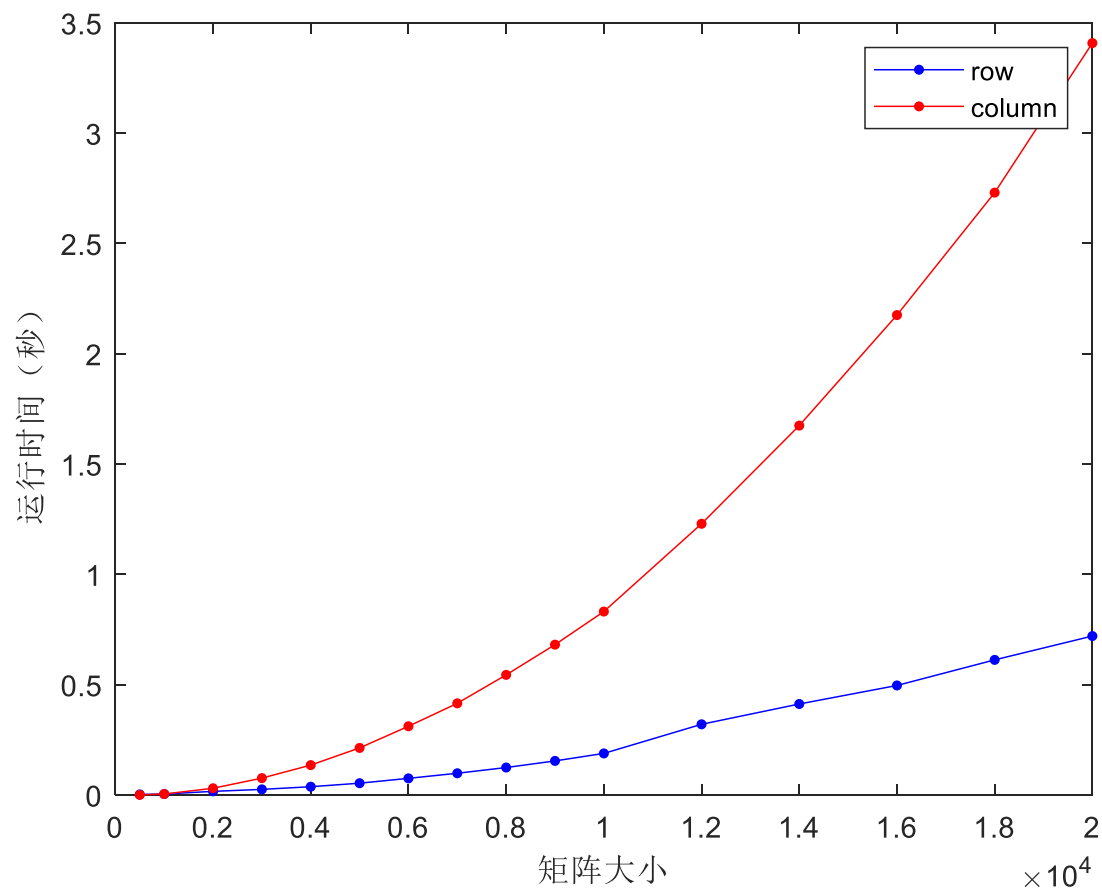
现统计不同次数情况下行列优先耗时的平均结果：



从图中可看出按行优先顺序执行二维数组幅值操作所耗时间稳定地低于按照列优先顺序的结果

Row-major order or column-major order?

现统计不同矩阵大小下行列优先耗时的平均结果：



从图中可看出按行优先顺序执行二维数组赋值操作所耗时间稳定地低于按照列优先顺序的结果

Row-major order or column-major order?

与其它语言的比较:

虽然C/C++采用行优先顺序原则，但像Fortran语言则采用列优先顺序，Matlab早期作为Fortran库的封装，因此其采用了列优先原则。这给我们的启示是在选择不同编程语言操作多维数组时，应遵循他们各自的行列优先原则，从而提高计算效率。

```

1  #include <stdio.h>
2
3  int main() {
4      printf("Input a string with space: ");
5      char str1[100], str2[100];
6      gets(str1);
7      char c;
8      char n;
9      scanf("%c", &c);
10     // n = getchar();
11     gets(str2);
12
13     printf("c = %d\n", c);
14     // printf("n = %d\n", n);
15     printf("str1 = %s, str2 = %s\n", str1, str2);
16     return 0;
17
18 }

```

这段代码的作用是先读入一行字符串给str1，再读入一个字符给c，然后再读入一个字符给str2。

run L09_input.c

```

C:\Users\night\Desktop\新文1 >
Input a string with space: i have a pen
a
c = 97
str1 = i have a pen, str2 =

-----
Process exited after 4.872 seconds with return value 0
请按任意键继续. . .

```

程序跳过了str2的读取

```
1 #include <stdio.h>
2
3 int main() {
4     printf("Input a string with space: ");
5     char str1[100], str2[100];
6     gets(str1);
7     char c;
8     char n;
9     scanf("%c", &c);
10    n = getchar();
11    gets(str2);
12
13    printf("c = %d\n", c);
14    printf("n = %d\n", n);
15    printf("str1 = %s, str2 = %s\n", str1, str2);
16    return 0;
17 }
18 }
```

我们发现，scanf读取完字符c后，留了一个换行符，如果直接读取str2会给str2直接赋值换行符从而跳过真正的str2输入

因此，使用scanf输入一个字符型后，使用一个getchar()将多余的换行符读掉，可以不保存到变量中。

```
C:\Users\night\Desktop\新文1 >
Input a string with space: i have a pen
a
i have an apple
c = 97
n = 10
str1 = i have a pen, str2 = i have an apple

-----
Process exited after 12.82 seconds with return value 0
请按任意键继续. . .
```

ASCII值	控制字符
0	NUL
1	SOH
2	STX
3	ETX
4	EOT
5	ENQ
6	ACK
7	BEL
8	BS
9	HT
10	LF
11	VT
12	FF
13	CR
14	SO

scanf() and printf()

```
scanf("%c %c %c",&a,&b,&c);
```

```
scanf(" %c %c %c",&a,&b,&c);
```

???

```
scanf(" %c %c %c ",&a,&b,&c);
```

```
scanf(" %c, %c %c",&a,&b,&c);
```

```
scanf(" %c , %c %c",&a,&b,&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()
{
```

```
    // 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;
```

```
}
```

```
int a = b + c;
```

```
Sorting array
```

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

```
int main()  
{
```

// face detection

// face mask detection

// face recognition

// decision making

return 0;

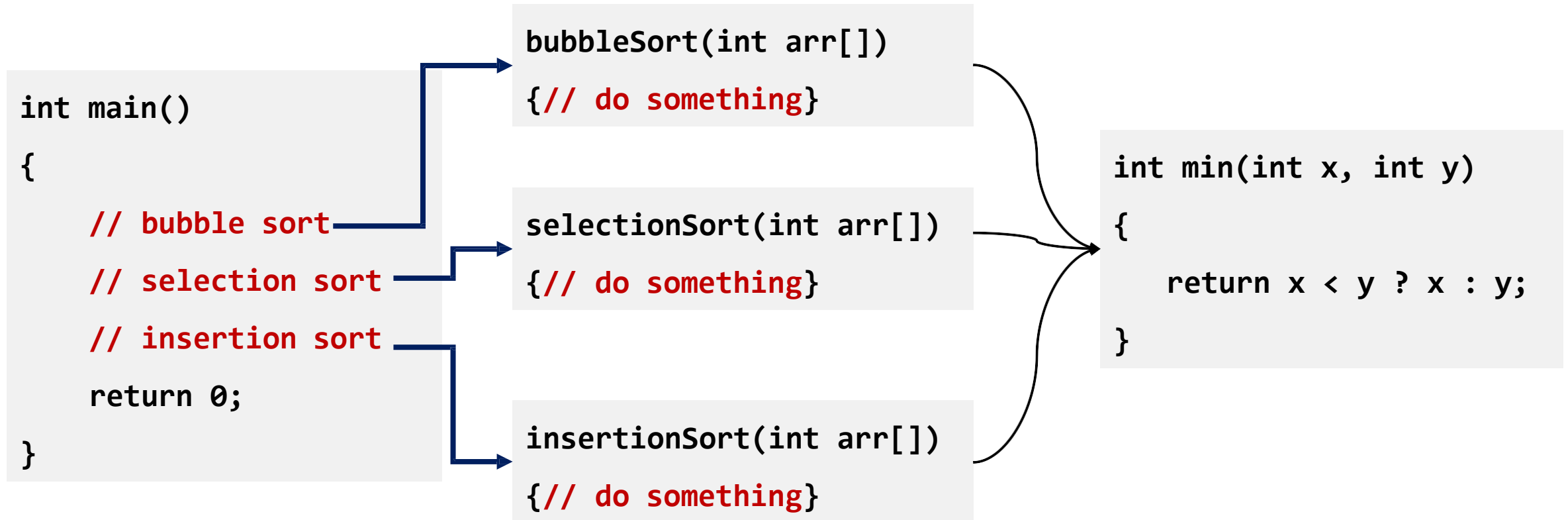
```
}
```

```
int recogFace()  
{ // 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;
}
```

`include<xxx.h>`

Self-defined function

Declare a function (声明)



Define a function (定义)



Call a function (调用)

How to separate functions

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

```
main()
```

```
{
```

```
    int x = 20, y = 10;
```

```
    int z = x + y; _____
```

```
    int max = x > y ? x:y; _____
```

```
}
```

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

```
{
```

```
    int z = x + y;
```

```
    return z;
```

```
}
```

```
int max(int x, int y)
```

```
{
```

```
    int z = x > y ? x : y;
```

```
    return z;
```

```
}
```

How to separate functions

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

```
main()
```

```
{
```

```
    int x = 20, y = 10;
```

```
    int z = x + y;
```

```
    int max = x > y ? x : y;
```

```
}
```

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

```
{
```

```
    return x + y;
```

```
}
```

You can directly return the result!

```
int max(int x, int y)
```

```
{
```

```
    return x > y ? x : y;
```

```
}
```

Function

①

Declare 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;
```

```
}
```

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

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

```
main()
```

```
{
```

```
    int x = 20, y = 10;
```

```
    int z = max(x, y);
```

```
}
```

```
int max(int x, int y)
```

```
{
```

```
    return x > y ? x : y;
```

```
}
```

③

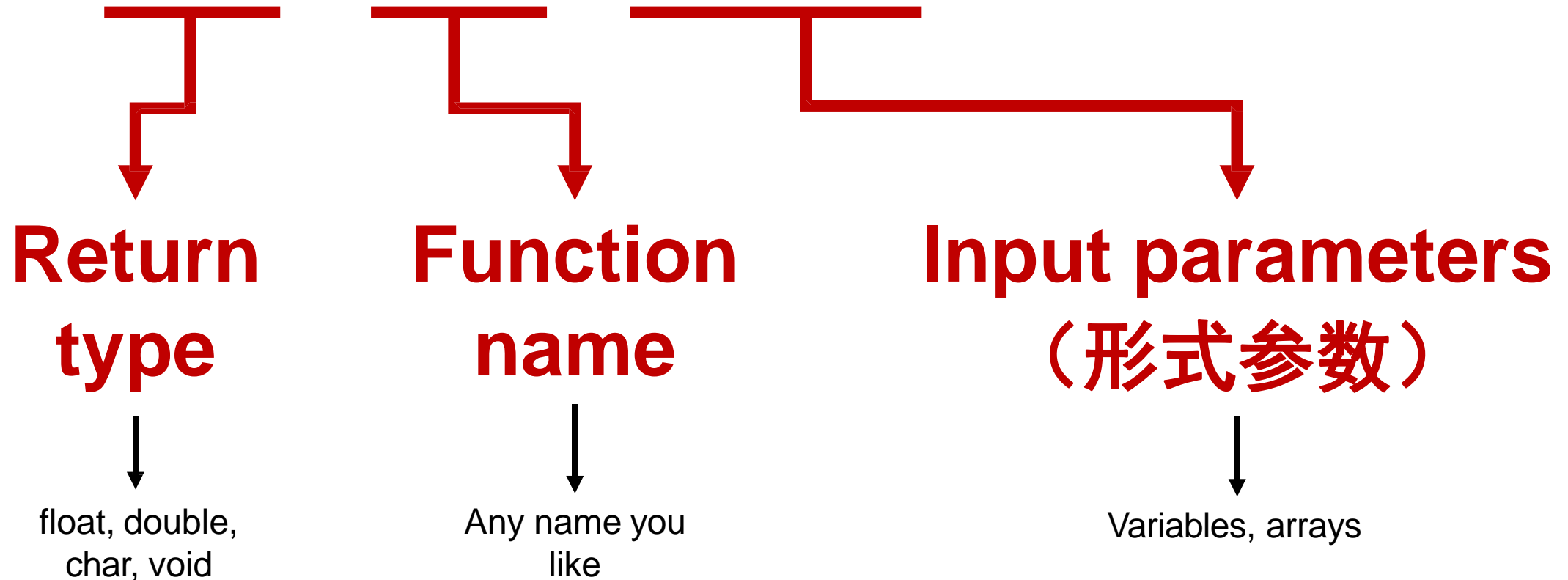
Call function

②

Define function

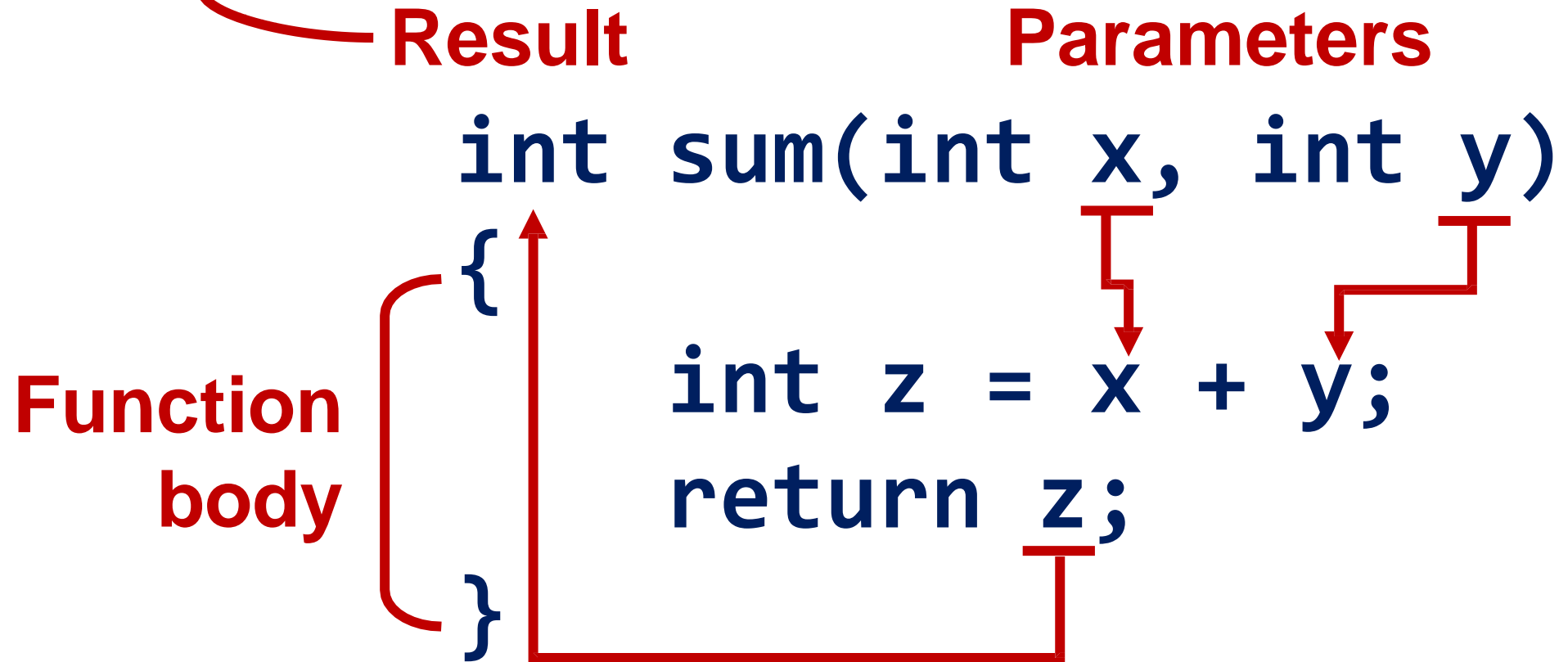
Declare a function

int sum(int x, int y);



Define a function

main



Define a function

在被定义的函数中，必须指定形参的类型。

Parameters

```
sum(int x, int y)
{
    int z = x + y;
    printf("x+y=%d", z);
}
```

Function body

Call a function

```
main()
```

```
{
```

```
    int x = 20, y = 10;
```

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



```
}
```

Arguments
(实际参数)

Function positioning matters

①

**Declare and
define function
before main!!!**

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

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

```
main()
```

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

```
#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);
}
```

②

Call function

Function positioning matters

```
#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

```
#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;  
}
```

Function positioning matters

函数的“定义”和“声明”的区别：

- 函数的定义是指对函数功能的确立，包括指定函数名，函数值类型、形参及其类型、函数体等，它是一个完整的、独立的函数单位。
- 函数的声明的作用则是把函数的名字、函数类型以及形参的类型、个数和顺序通知编译系统，以便在调用该函数时系统按此进行对照检查。

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;}`

在定义函数中指定的形参，在未出现函数调用时，它们并不占内存中的存储单元。只有在发生函数调用时，形参才被分配内存单元。在调用结束后，形参所占的内存单元也被释放。

Definition must be consistent

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

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

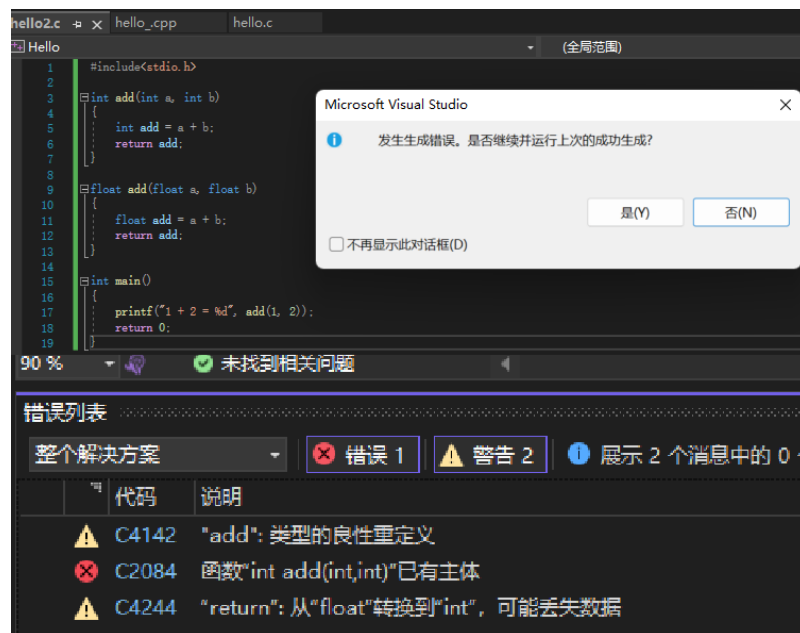
```
float add(float a, float b)
{
    float add = a + b;
    return add;
}
```

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

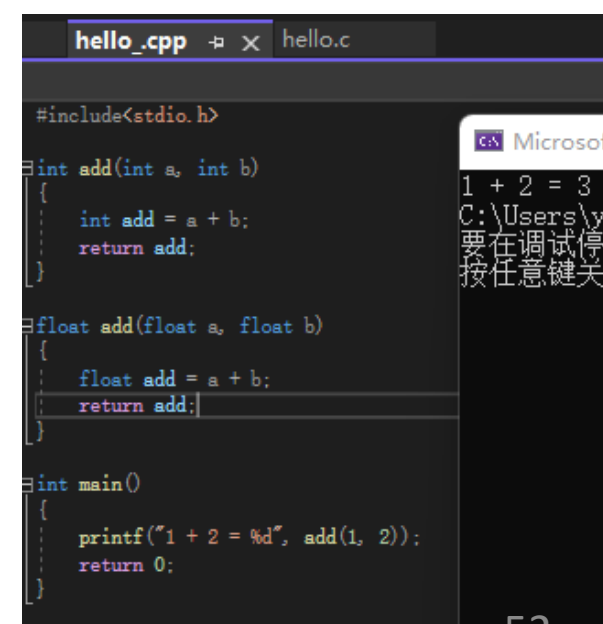
C1x:泛型选择
(高级用法)

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

C



C++



Definition must be consistent

在定义函数时指定的函数类型一般应该和return语句中的表达式类型一致。

- 如果函数值的类型和return语句中表达式的值不一致，则以**函数类型为准**。
- 对数值型数据，可以自动进行类型转换。即**函数类型决定返回值的类型**。

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

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

function and array

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

```
int sum_array(int a[], int n);
```

```
int sum_array(int [], int n);
```

```
int sum_array(int [], int );
```

```
int sum_array(int a[], int n)
```

```
{  int i, sum=0;
```

```
    for(i=0; i<n; i++)
```

```
        sum+=a[i];
```

```
    return sum;
```

```
}
```

Declare function
(All are OK)



```
int main(void){  
...  
total=sum_array(b, 50);  
...  
}
```

Arguments and parameters

C有2种参数传递方式:

- ✓ 值传递
- ✓ 地址传递

Value

System
creates a new
memory unit

Arguments
(实参)

实参与形参的类型应
相同或赋值兼容。

Parameters
(形参)

Address

System uses
the existing
memory unit

Arguments and parameters

```
#include <stdio.h>

int sum(int x, int y);

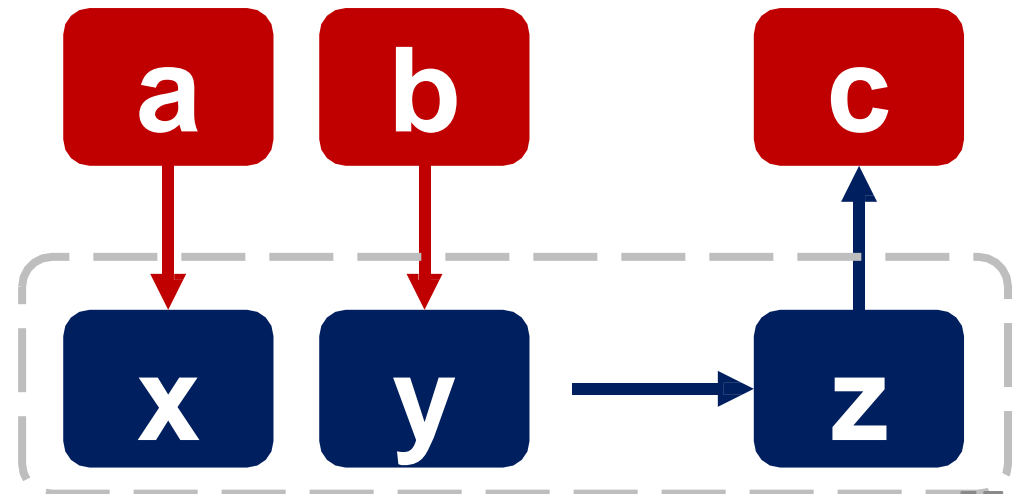
int main(void)
{
    int a = 100;
    int b = 200;

    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);

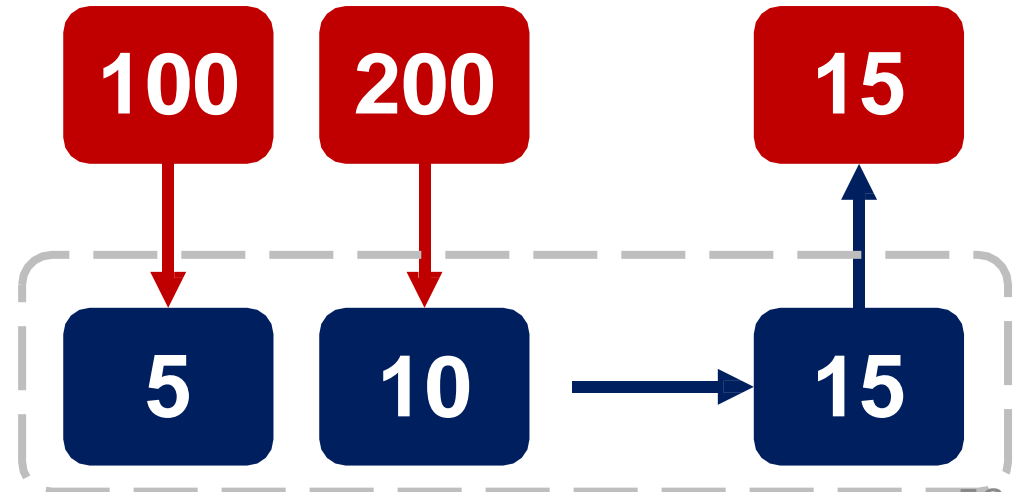
main ()
{
    int a = 100;
    int b = 200;

    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;

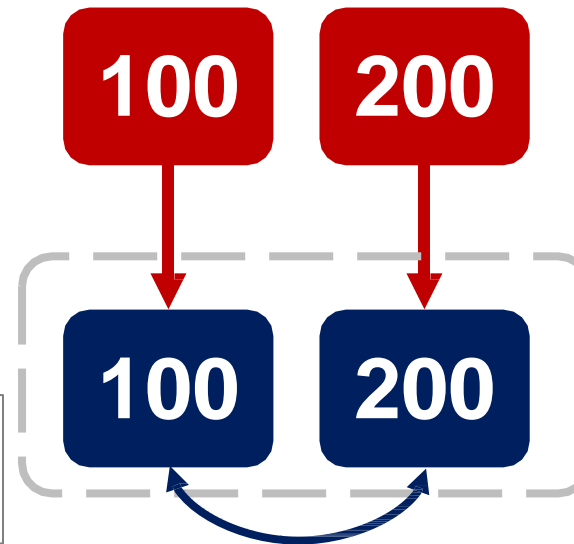
    swap(a, b);
}
```

**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;
}
```

run
L09_swap.cpp



**Can a and b
be swapped?**

实参可以是**常量**、**变量**或**表达式**，但要求它们有确定的值。在调用时将实参的值赋给形参。

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;
}
```

run
[L09_swap2.cpp](#)

**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 (int* array or int array[])**

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)
```

```
{
```

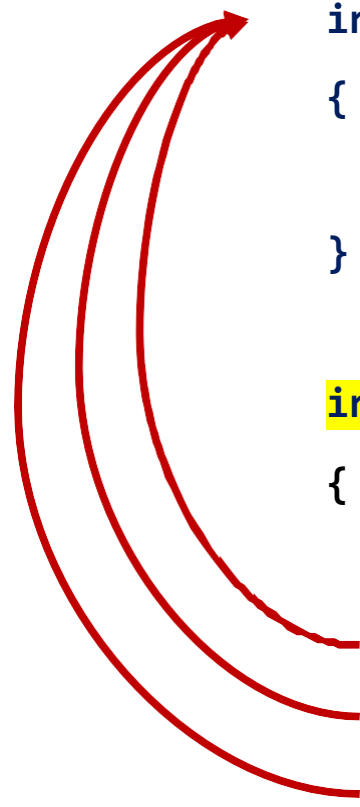
```
    int z;
```

```
    z = max(a, b);
```

```
    z = max(z, c);
```

```
    z = max(z, d); return z;
```

```
}
```



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

函数间的调用关系:由主函数调用其他函数，其他函数也可以互相调用。同一个函数可以被一个或多个函数调用任意多次。

Functions can be nested

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

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

```
int sum(int a[], int len);
```

```
main()
```

```
{
```

```
    int a[] = {1, 3, 4, 5, 2, 9, -3, 2};
```

```
    int z = sum(a, sizeof(a) / sizeof(a[0]));
```

```
}
```

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

```
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;
```

```
}
```

Functions can be mutually nested

```
#include <stdio.h>

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

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

    sum(a, b);
}

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

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

The diagram illustrates mutual recursion between the `sum` and `sub` functions. A red arrow originates from the `sum(a, b);` call inside the `main` function and points to the `sum` function definition. Another red arrow starts from the `sub(a, b);` call inside the `sum` function definition and points to the `sub` function definition. A third red arrow starts from the `sum(a, b);` call inside the `sub` function definition and points back to the `sum` function definition. These arrows form a cycle, demonstrating how the functions can call each other indefinitely, leading to a dead loop.

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

Function can be self-called

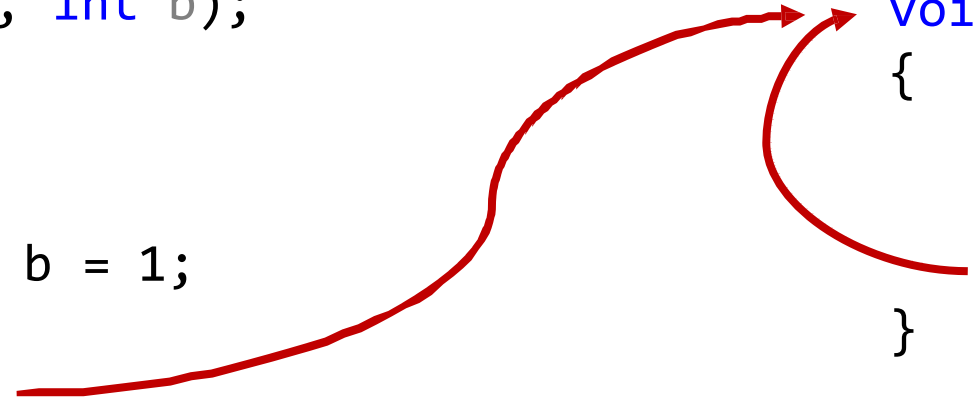
```
#include <stdio.h>

void sum(int a, int b);

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

    sum(a, b);
}

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



This is recursion!!!

Summary

如果实参表列包括多个实参，对实参求值的顺序并不是确定的，有的系统按自左至右顺序求实参的值，有的系统则按自右至左顺序。

```
int i=2, p;
```

```
p=f(i, ++i);
```

如果按自左至右顺序求实参的值，则函数调用相当于f(2,3)

如果按自右至左顺序求实参的值，则函数调用相当于f(3,3)

Summary

按函数在程序中出现的位置来分，可以有以下三种函数调用方式：

- **函数语句**。把函数调用作为一个语句。这时不要求函数带回值，只要求函数完成一定的操作。`printf("%n",i);`
- **函数表达式**。函数出现在一个表达式中，这种表达式称为**函数表达式**。这时要求函数带回一个确定的值以参加表达式的运算。例如：`c=2*max(a,b);`
- **函数参数**函数调用作为一个函数的实参

`m = max (a , max (b , c)) ;`

其中`max (b , c)`是一次函数调用，它的值作为`max`另一次调用的实参。

`m`的值是`a`、`b`、`c`三者中的最大者。

Summary

(1) 一个 C 程序由一个或多个程序模块组成，每一个程序模块作为一个源程序文件。对于较大的程序，通常将程序内容分别放在若干个源文件中，再由若干源程序文件组成一个 C 程序。这样便于分别编写、分别编译，提高调试效率。一个源程序文件可以为多个 C 程序公用。

(2) 一个源程序文件由一个或多个函数以及其他有关内容（如命令行、数据定义等）组成。一个源程序文件是一个编译单位，在程序编译时是以源程序文件为单位进行编译的，而不是以函数为单位进行编译的。

Summary

(3) C 程序的执行是从main函数开始的，如果在main函数中调用其他函数，在调用后流程返回到main函数，在main函数中结束整个程序的运行。

(4)所有函数都是平行的，即在定义函数时是分别进行的，是互相独立的。一个函数并不从属于另一函数，即函数不能嵌套定义。函数间可以互相调用，但不能调用main函数。main函数是系统调用的。

Summary

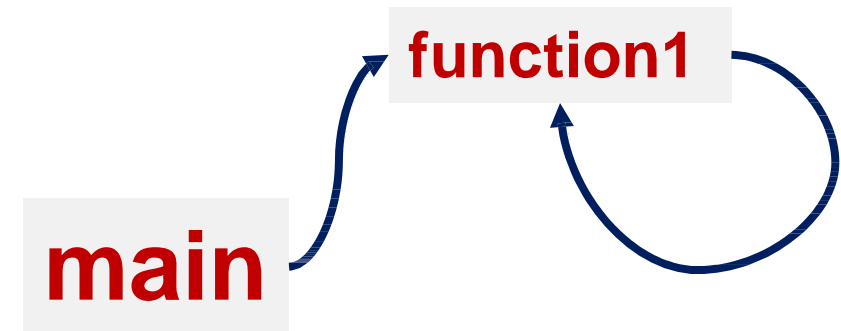
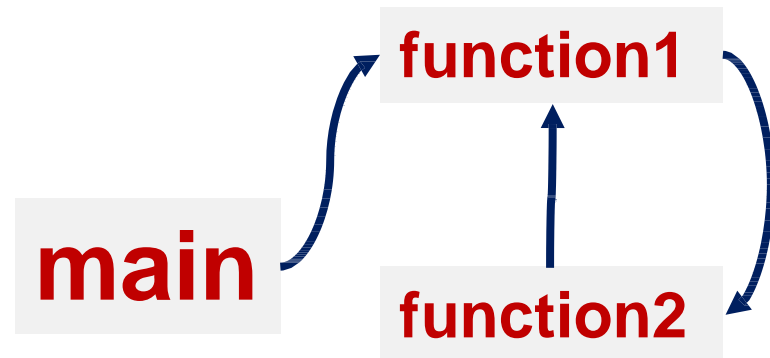
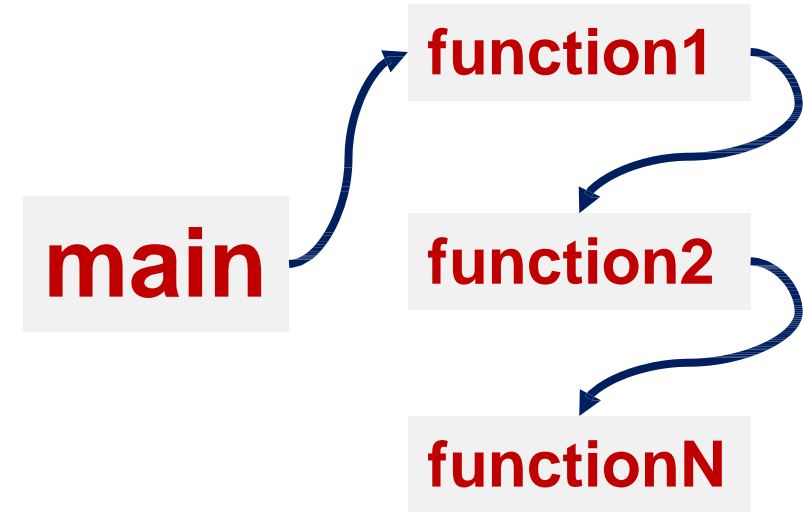
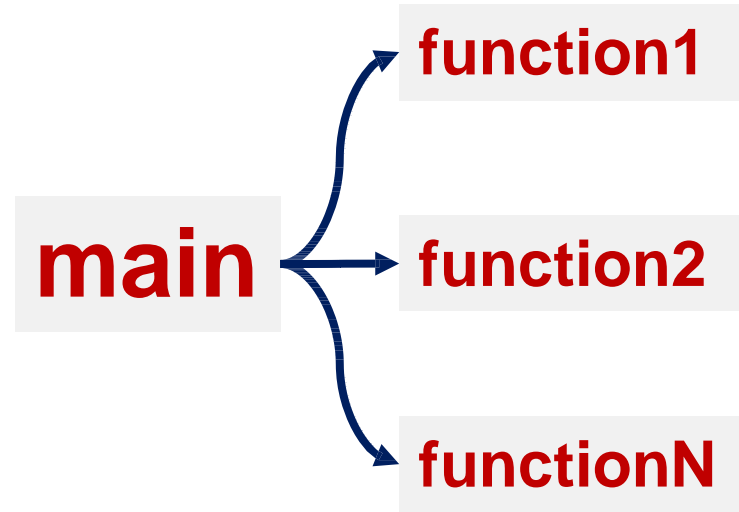
(5) 从用户使用的角度看，函数有两种：

- ① 标准函数，即**库函数**。由系统提供的，用户不必自己定义这些函数，可以直接使用它们。不同的C系统提供的库函数的数量和功能会有一些不同，但许多基本的函数是共同的。还应该在本文件开头用**#include**命令将调用有关库函数时所需用到的信息“包含”到本文件中来。
- ② **用户自己定义的函数**。用以解决用户的专门需要。

(6) 从函数的形式看，函数分两类：

- ① **无参函数**。无参函数一般用来执行指定的一组操作。在调用无参函数时，主调函数不向被调用函数传递数据。
- ② **有参函数**。主调函数在调用被调用函数时，通过参数向被调用函数传递数据。

Functions



Case study: insertion sort

```
#include<stdio.h>
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);

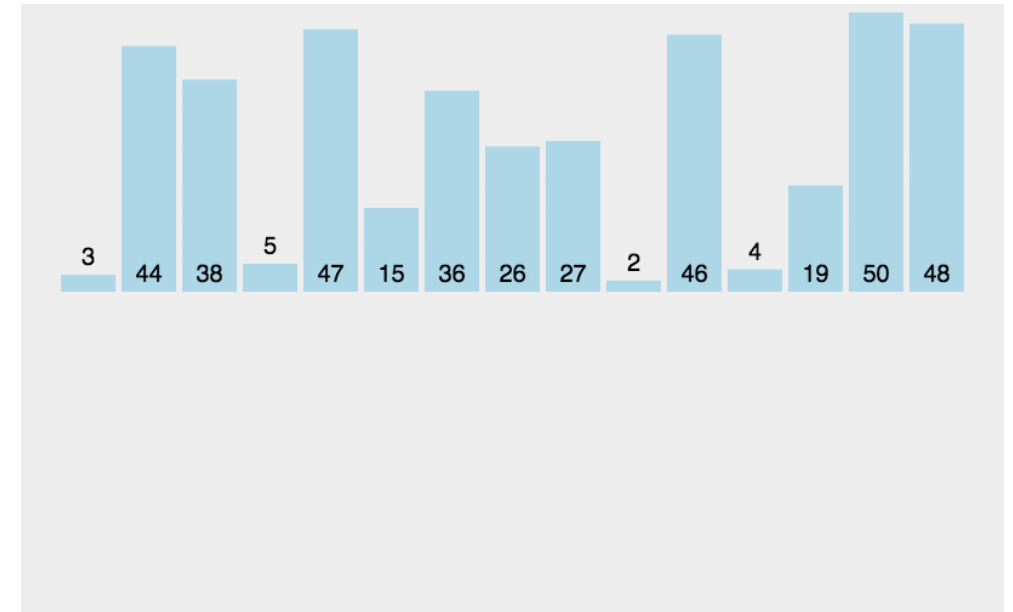
    insertion_sort(arr, len);

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

Define function

Call function

Case: create insertion sort algorithm!

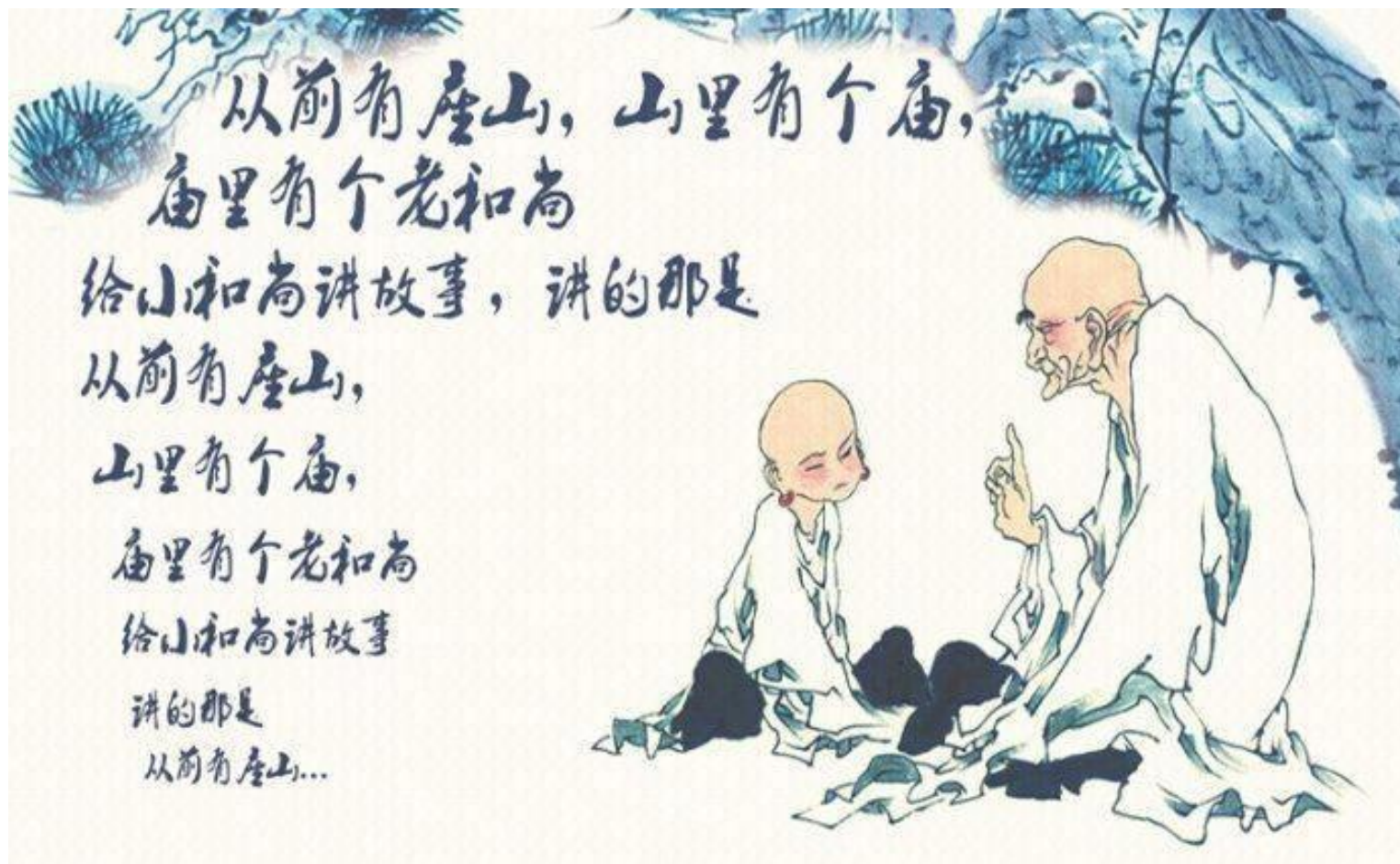


Microsoft Visual Studio Debug Console

```
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!

[illegible]

Case study: 从前有座山…

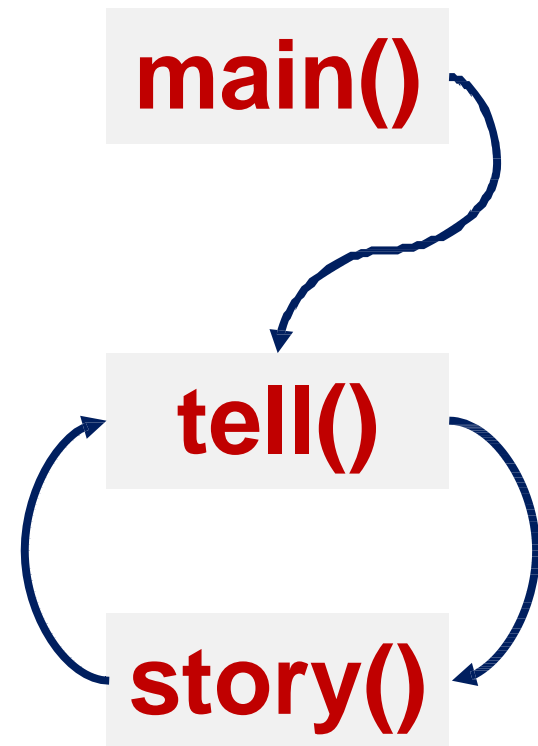
```
#include <stdio.h>

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

```
#include<stdio.h>
int fever(float *tems, int size);
int main(void)
{
    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!



C:\MICROSOFT VISUAL STUDIO

有 1 个人发烧了
C:\Users\vdf19\source

Case study: find 3 nearest points

```
#include<stdio.h>
float dist(int x1, int y1, int x2, int y2);

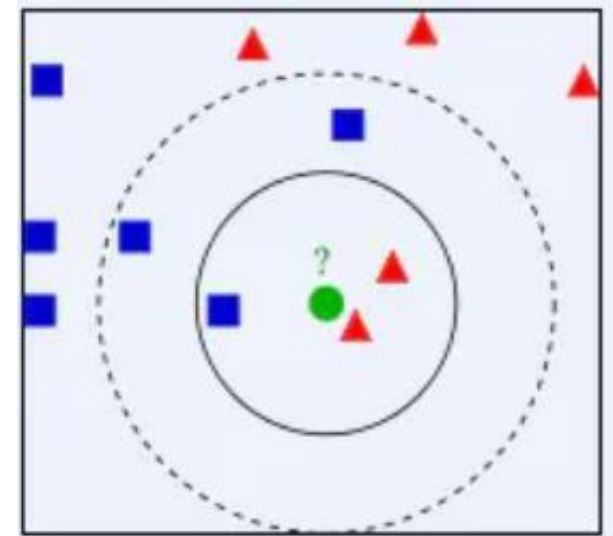
int main(void)
{
    int pts[5][2] = {{2,54},{45,67},{25,5},{23,62},{86,34}};
    int center[2] = { 25,25 };
    float D[5] = {};
    for (int i = 0; i < 5; i++)
        D[i] = dist(center[0],center[1],pts[i][0],pts[i][1]);

    insertion_sort(D, 5);

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

float dist(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
```

Case study: calculate the average amount

```
#include <stdio.h>

int main(void) {
    float average(float array[10]); /*函数声明 */
    float score[10], aver;
    int i;
    printf("input 10 scores: \n");
    for (i = 0; i < 10; i++)
        scanf("%f", &score[i]);
    printf("\n");
    aver = average(score);
    printf("average score is %5.2f\n", aver);
}
```

```
float average (float array[10]) {
    int i ;
    float aver, sum = array[0];
    for (i = 1; i < 10; i++)
        sum = sum + array[i];
    aver = sum / 10;
    return (aver);
}
```

 C:\Users\12096\Desktop\try.exe

```
input 10 scores:
100 56 78 98.5 76 87 99 67.5 75 9
average score is 83.40
```

Case study: calculate the average amount

形参数组可以不定义长度

```
#include <stdio.h>

int main(void) {
    float average(float array[], int n);
    float score_1[5] = {98.5, 97, 91.5, 60, 55};
    float score_2[10] = {67.5, 89.5, 99, 69.5, 77, 89.5, 76.5, 54, 60, 99.5};
    printf("the average of class A is %6.2f\n", average(score_1, 5));
    printf("the average of class B is %6.2f\n", average(score_2, 10));
    return 0;
}

float average(float array[], int n) {
    int i;
    float aver, sum = array[0];
    for (i = 1; i < n; i++)
        sum = sum + array[i];
    aver = sum / n;
    return aver;
}
```

C:\Users\12096\Desktop\try.exe

```
the average of class A is 80.40
the average of class B is 78.20
```

Content

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

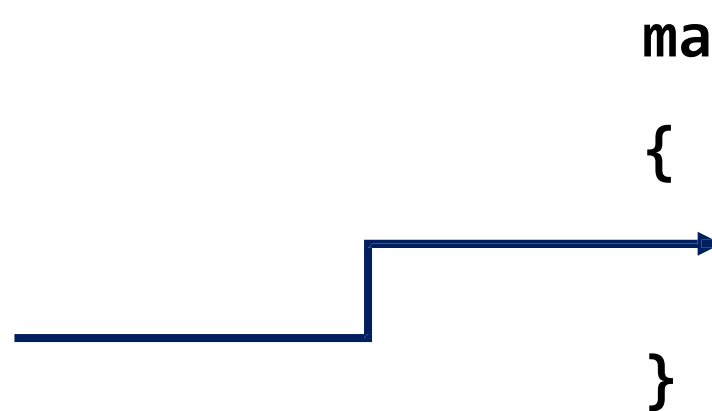
Variable scope

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)

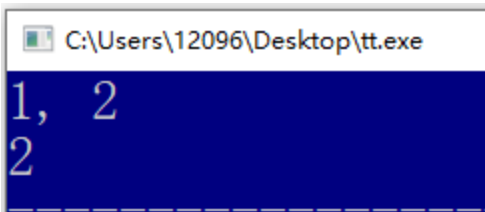
 `int b; //outside main`

Local variable
(局部变量)

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

Variable scope

Global variable can be accessed everywhere



```
C:\Users\12096\Desktop\tt.exe
1, 2
2
```

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

```
int b = 2; // global
```

```
void func()
```

```
{
```

```
    printf(" %d", b);
```

```
}
```

```
int main()
```

```
{
```

```
    int a = 1; // local
```

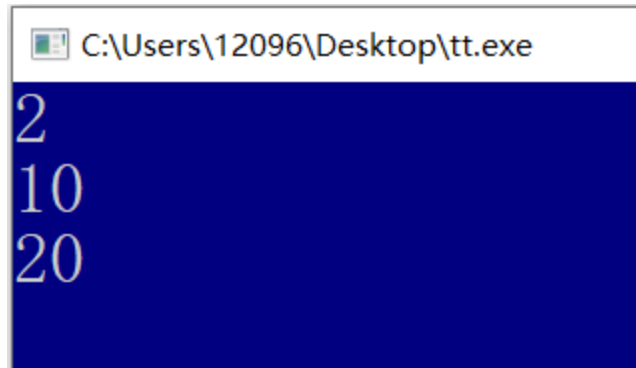
```
    printf("%d, %d", a, b);
```

```
    func(); return 0;
```

```
}
```

Variable scope

Global variable can be changed everywhere and keep the changes



```
C:\Users\12096\Desktop\tt.exe
2
10
20
```

```
int b = 2; // global
void func()
{
    printf("%d\n", b); //print 10
    b = 20;
}
int main()
{
    printf("%d\n", b); //print 2
    b = 10;
    func();
    printf("%d\n", b); //print 20
    return 0;
}
```

Variable scope

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

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

```
int b = 2; // global
```

```
void func() {
```

```
    printf("%d", b); //print 2
```

```
    b = 20;
```

```
}
```

```
int main() {
```

```
    int b = 5; // local
```

```
    printf("%d", b); //print 5
```

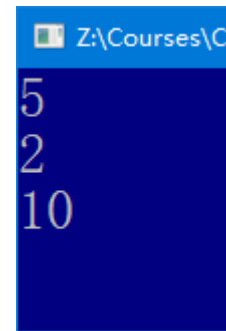
```
    b = 10;
```

```
    func();
```

```
    printf("%d", b); //print 10
```

```
    return 0;
```

```
}
```



Variable scope

```
float PI = 3.14; // global

float func(float a)
{
    return a * PI;
}

main()
{
    float a = 5; // local

    float b = func(a);
    printf("%f", b);
}
```

- ✓ **Do not use global variables unless**
 - It is a **constant** that can be used everywhere (consensus一致性)
 - 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;`

常用

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;
}
```

Both i and j are auto variables

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

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
}
```

Static variable

(1) 静态局部变量属于静态存储类别，在静态存储区内分配存储单元。**在程序整个运行期间都不释放**。而自动变量（即动态局部变量）属于动态存储类别，占动态存储区空间而不占静态存储区空间，**函数调用结束后即释放**。

(2) 对静态局部变量是在编译时赋初值的，即**只赋初值一次**，在程序运行时它已有初值。以后每次调用函数时不再重新赋初值而只是保留上次函数调用结束时的值。而对自动变量赋初值不是在编译时进行，而是在函数调用时进行，每调用一次重新给一次初值，相当于执行一次赋值语句。

Static variable

(3) 如在定义局部变量时不赋初值的话，则对静态局部变量来说，编译时自动赋初值 0（对数值型变量）或 '\0'（对字符变量）。

而对自动变量来说，如果不赋初值则它的值是一个不确定的值。

(4) 虽然静态局部变量在函数调用结束后仍然存在，但其他函数不能引用它。

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;    // 定义外部变量  
int num = 30;  
char str[81] = "abcdefg";
```

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

```
/**> File Name: main_test.c***/  
#include <stdio.h>  
//将已定义的外部变量的作用域扩展到本文件  
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;
```

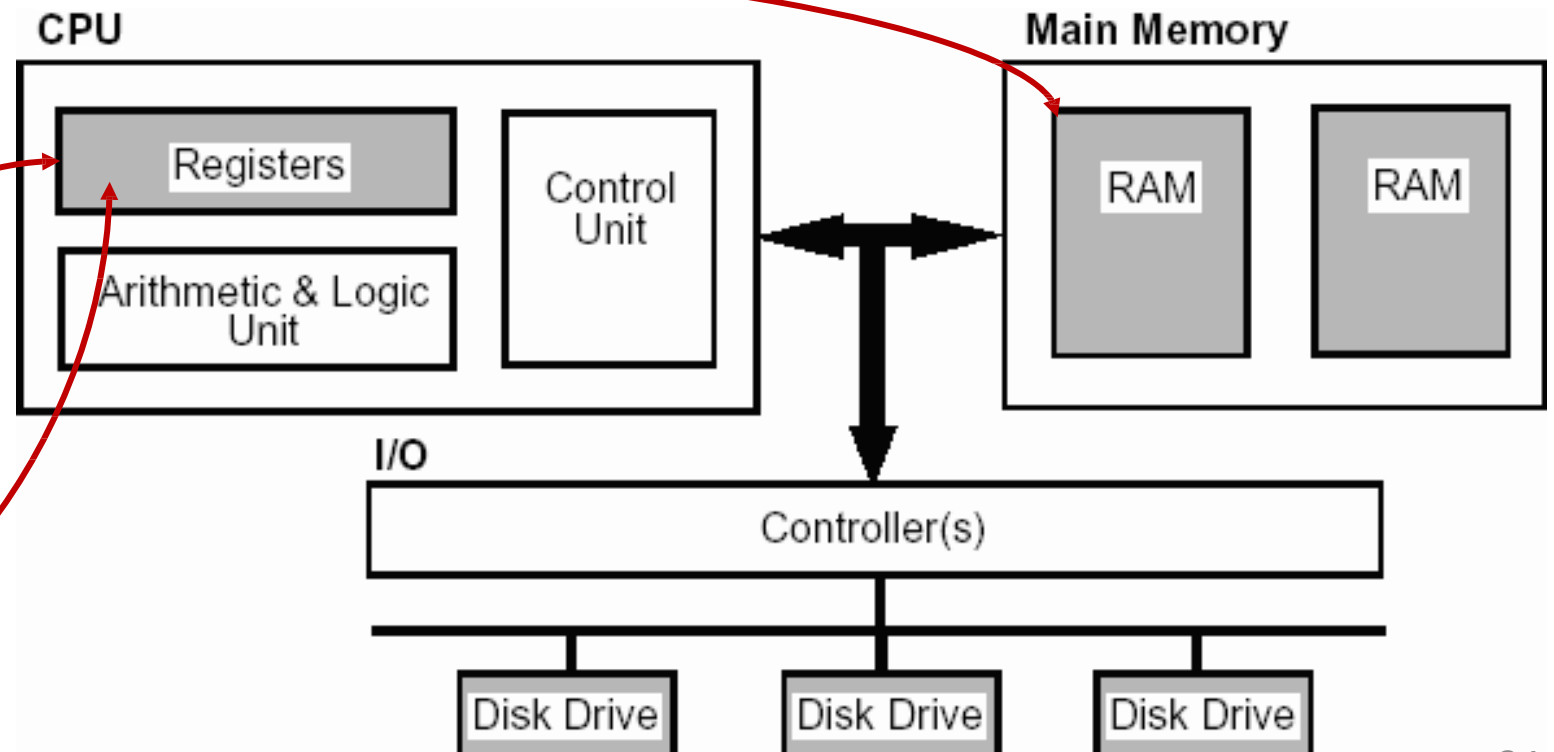
```
    register int d = 2;
```

```
}
```

```
main(){
```

```
    int a = 10;
```

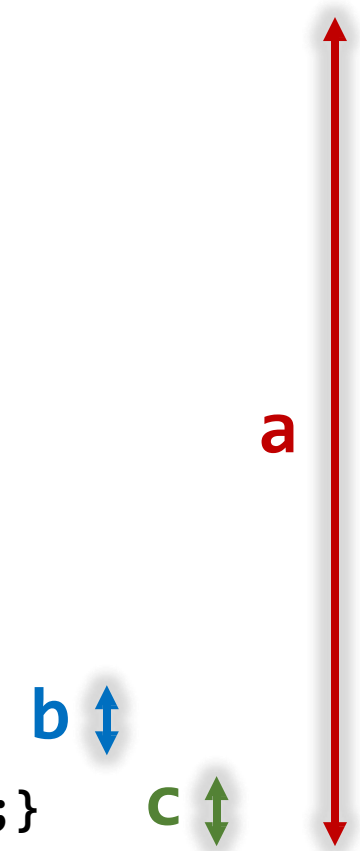
```
}
```



Variable scope

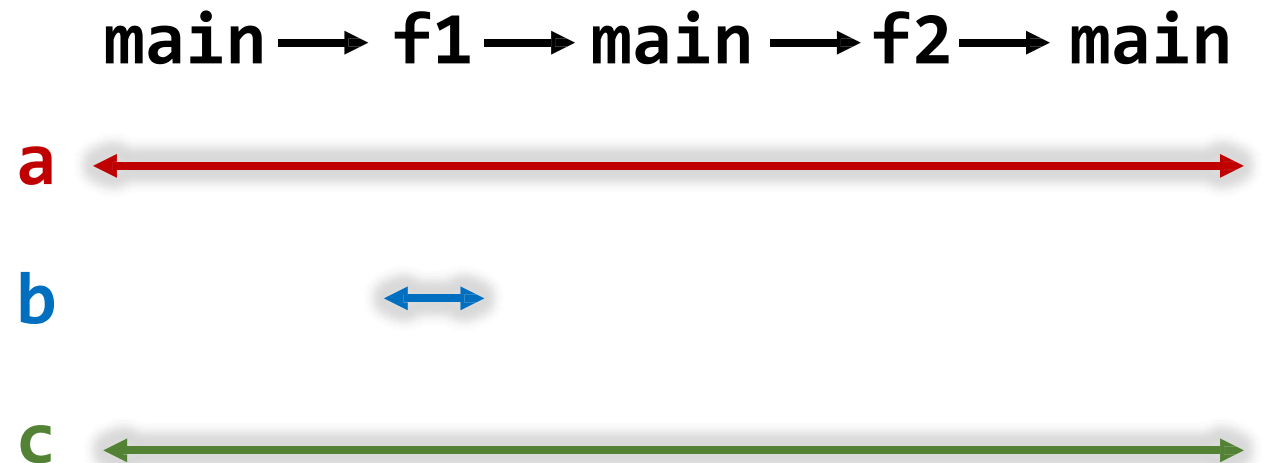
Scope in space

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



The diagram illustrates the spatial scope of variables. A long red vertical double-headed arrow labeled 'a' spans the entire height of the code, indicating that variable 'a' is in global scope. A blue vertical double-headed arrow labeled 'b' is positioned next to the 'f1()' function definition, indicating its local scope. A green vertical double-headed arrow labeled 'c' is positioned next to the 'f2()' function definition, indicating its local scope.

Scope in time



Summary

(1) 从作用域角度分，有局部变量和全局变量。它们采用的存储类别如下：

- 局部变量包括：
自动变量、静态局部变量、寄存器变量。
形式参数可以定义为自动变量或寄存器变量。
- 全局变量包括：
静态外部变量、外部变量。

Summary

- (2) 从变量存在的时间来区分，有动态存储和静态存储两种类型。
- 静态存储是程序整个运行时间都存在，而动态存储则是在调用函数时临时分配单元。
- 动态存储：自动变量、寄存器变量、形式参数。
 - 静态存储：静态局部变量、静态外部变量、外部变量。

Summary

(3) 从变量值存放的位置来区分,可分为:

内存中静态存储区: 静态局部变量、静态外部变量、外部变量。

内存中动态存储区: 自动变量和形式参数。

CPU中的寄存器: 寄存器变量。

(4) `static`对局部变量和全局变量的作用不同。对局部变量来说,它使变量由动态存储方式改变为静态存储方式。而对全局变量来说,它使变量局部化,但仍为静态存储方式。从作用域角度看,凡有`static`声明的,其作用域都是局限的,或者是局限于本函数内,或者局限于本文件内。

Case study: static variable

```
#include <stdio.h>

void func(void);

static int count = 5;
int main(void) {

    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).

 Microsoft Visual Studio Debug Co

```
i is 6 and count is 4
i is 7 and count is 3
i is 8 and count is 2
i is 9 and count is 1
i is 10 and count is 0
```


Content

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

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();  
}
```

Recursive call

Function call

Recursion

Recursively subtract

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

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

A blue line starts from the `recurse(n);` call in `main()`, goes right, then up, then left to point at the `recurse(n-1);` line in the `recurse` function. A red **100** is placed to the right of the `recurse(n-1);` line. A red **99, 98, 97, ..., 0, -1, ...** is placed below the `recurse(n-1);` line.

Recursively add

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

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

A blue line starts from the `recurse(n);` call in `main()`, goes right, then up, then left to point at the `recurse(n+1);` line in the `recurse` function. A red **0** is placed to the right of the `recurse(n+1);` line. A red **1, 2, 3, ..., 100, 101, ...** is placed below the `recurse(n+1);` line.

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

```
#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

```
#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.

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

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

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 汉诺塔



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，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:

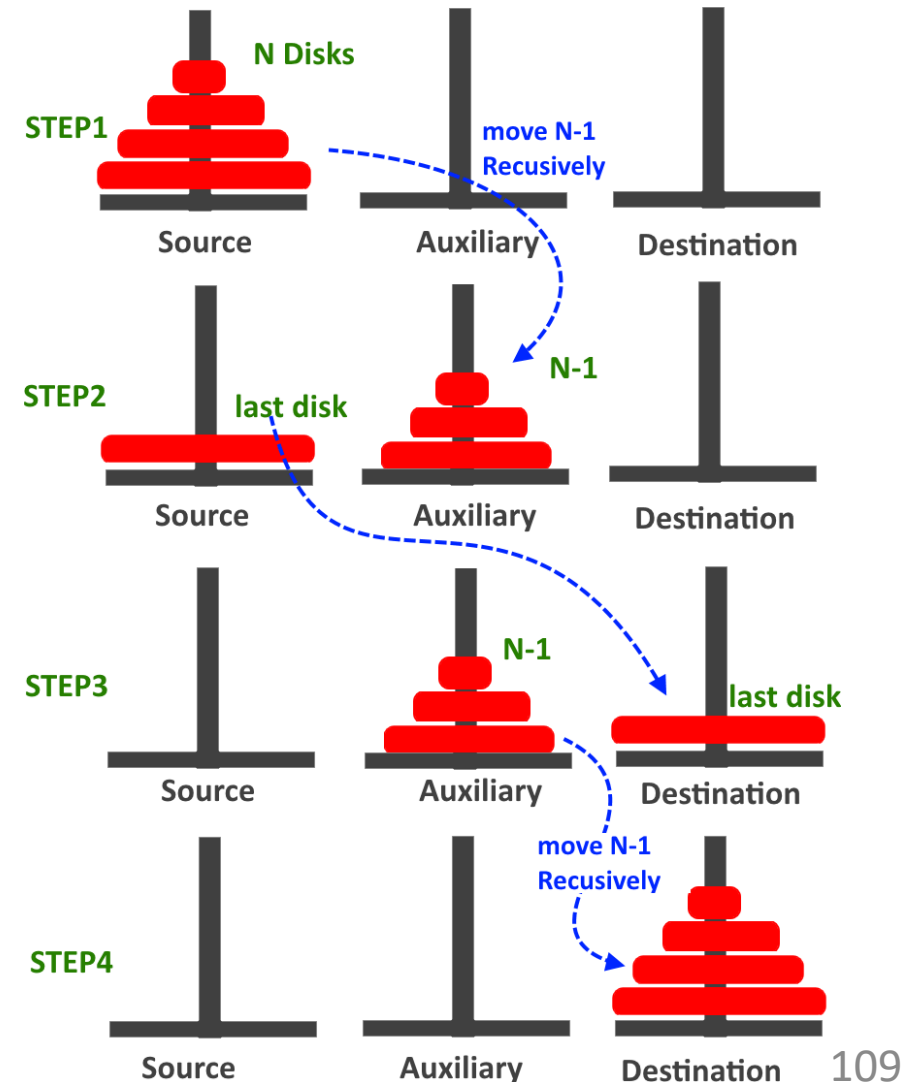
1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
3. No disk can be placed on top of a disk that is smaller than it.

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

```
#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

```
#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

- 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.