函数

2022年3月18日 9:51

数学上的函数：必须有参数，必须有返回值，并且不会产生副作用

C语言中的函数：可以没有参数，也可以没有返回值，甚至可能会在函数调用中产生副作用

函数的定义

```
return_type  func_name(parameter list) {
    statements  ——> 函数体
}
```

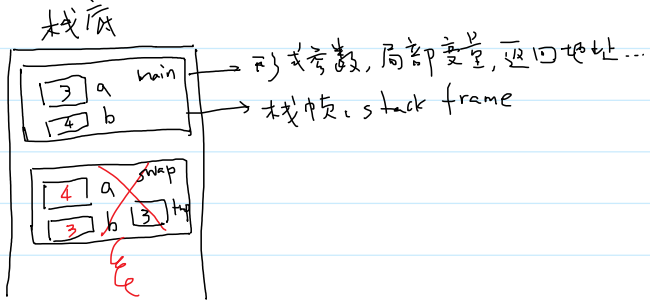
#1. 参数是如何传递的？（值传递）（★★★）

↓

C语言中只有值传递

```c
#include <stdio.h>

void swap(int a, int b);

int main(void) {
    int a = 3, b = 4;
    printf("a = %d, b = %d\n", a, b);
    swap(a, b);
    printf("a = %d, b = %d\n", a, b);

    return 0;
}

void swap(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}
```

```
a = 3, b = 4
a = 3, b = 4
```



如何在被调函数中修改调用函数的值？（指针）

1）一个函数把作为参数（退化为指针）

```c
int main(void) {
    int arr[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    printf("sum = %d\n", sum_array(arr));

    return 0;
}

int sum_array(int arr[]) {
    int sum = 0;
    for (int i = 0; i < SIZE(arr); i++) {
        sum += arr[i];
    }
    return sum;
}
```

形式上是数组，但它其实是一个指针

① 数组作为参数传递会丢失类型信息，数组长度

数组作为参数传递时会退化为指向第一个元素的指针，

这么做有什么好处？

① 可以避免数组拷贝

② 可以修改原数组的值

③ 函数调用会更加灵活

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

int main(void) {
    int arr[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    printf("sum = %d\n", sum_array(arr, 5));

    return 0;
}

int sum_array(int arr[], int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += arr[i];
    }
    return sum;
}
```

求地址公式：i-addr = base_addr + i * sizeof(element_type)

不需要用到数组长度信息，因此可以省略

2) 二维数组作为参数传递

```c
int sum_array(int arr[][4], int n);

int main(void) {
    int matrix[3][4] = { {1, 2, 3, 4}, {2, 2, 3, 4}, {3, 2, 3, 4} };

    printf("sum = %d\n", sum_array(matrix, 3));

    return 0;
}

int sum_array(int matrix[][4], int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < 4; j++) {
            sum += matrix[i][j];
        }
    }
    return sum;
}
```

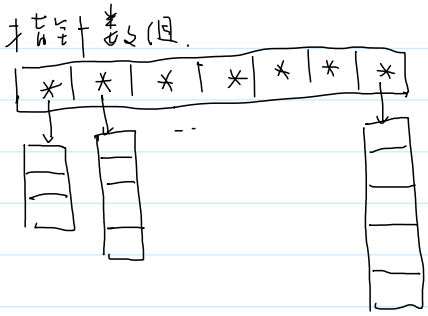
→ 地址公式 i-j-addr = base_addr + i * Columns * sizeof(element_type) + j * sizeof(element_type)

因此不能省略列

Q1: 为什么二维数组作为参数传递时，只能省略行的信息，不能省略列的信息.

Q2: 有没有办法传递一个列数不固定的二维数组呢？

指针数组.



#2 返回值.

注意事项：返回值类不能是数组.

#3. 程序如何终止.

操作系统会调用 main 函数 → 程序的开始

main 函数将状态码返回给操作系统 → 程序的终止.

Q. 如果不想在main 函数中终止程序，该怎么做？

exit

Defined in header <stdlib.h>

```
void exit( int exit_code );          (until C11)
```

```c
exit(EXIT_SUCCESS);
```

#define EXIT_SUCCESS 0

```
exit(EXIT_FAILURE);
```

#define EXIT_FAILURE 1

局部变量和外部变量

2022年3月18日 10:55

局部变量：定义在函数里面的变量。

外部变量（全局变量）：定义在函数外面的变量。

块作用域：{ } 从变量定义开始，到对应的块末尾。

文件作用域：从变量定义开始，到文件的末尾。

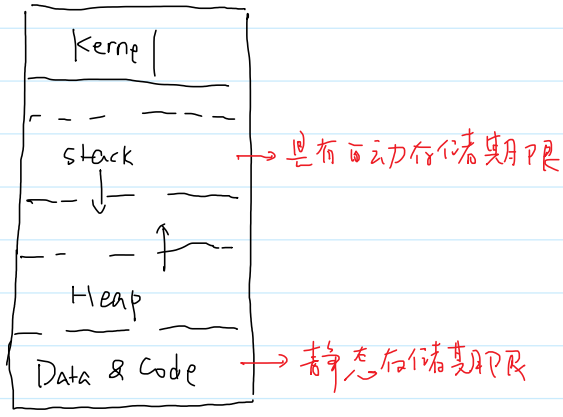
```c
int main(void) {
    int a = 3;
    printf("a = %d\n", a);

    for (int i = 0; i < 10; i++) {
        printf("a = %d\n", a);
        printf("i = %d\n", i);
    }

    printf("a = %d\n", a);
    printf("i = %d\n", i); → 又能够再访问了
}
```

```c
int main(void) {
    // printf("n = %d\n", n);
    foo();
    bar();

    return 0;
}

int n = 10;

void foo() {
    printf("n = %d\n", n);
}

void bar() {
    printf("n = %d\n", n);
}
```

1| 存储期限

自动存储期限：存放在栈里面的数据，具有自动存储期限，变量的生命周期随着栈帧的入栈而诞生，随着栈帧的出栈而结束。

静态存储期限：拥有永久的存储单元，在程序的整个执行期间都存在。




```c
void foo();

int main(void) {
    foo();
    printf("--------\n");
    foo();
    printf("--------\n");
    foo();
    return 0;
}

void foo() {
    int i;
    printf("i = %d\n", i++);
}
```

```c
void foo();

int main(void) {
    foo();
    printf("--------\n");
    foo();
    printf("--------\n");
    foo();
    return 0;
}

void foo() {
    static int i;
    printf("i = %d\n", i++);
}
```

```
i = 22067
--------
i = 22067
--------
i = 22067
```

```
i = 0
--------
i = 1
--------
i = 2
```

外部变量：静态存储。

内部变量：默认是自动存储期限，但是可以通过static关键字指定为静态存储期限。

递归（★★★★★）

2022年3月18日 11:36

recursion: re + cur + sion　　走重复的路

重复　走　名词后缀

递归：电影院的例子

1) Fibnacci 数列：0, 1, 1, 2, 3, 5, 8, 13, ...

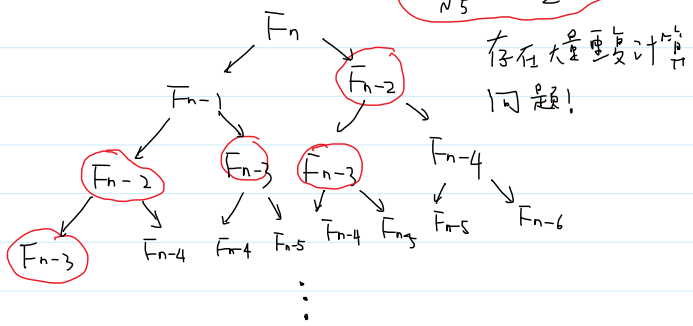
$$F_n = \begin{cases} F_{n-1} + F_{n-2}, & \text{if } n \geq 2 \\ 1, & n = 1 \\ 0, & n = 0 \end{cases}$$ （递归公式）

$$T_n = \begin{cases} 1, & n = 0 \\ 2, & n = 1 \\ T_{n-1} + T_{n-2} + 3, & n \geq 2 \end{cases}$$

$T_n \geqslant F_n$，递归树！

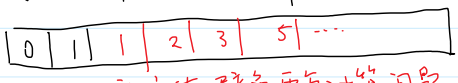
$$F_n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n \approx \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n$$

```
long long fib1(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fib1(n - 1) + fib1(n - 2);
}
```



存在大量重复计算问题！

Q：如何避免重复计算问题？



顺序求值避免重复计算问题。

动态规划。

```
long long fib2(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    int a = 0, b = 1;
    for (int i = 2; i <= n; i++) {
        int tmp = a + b; // 第i项的值
        // 为计算第i+1项做准备
        a = b;
        b = tmp;
    }
    return b;
}
```

$T_n = O(n)$

3) 能否用通项公式求解？ $F_n = \frac{1}{\sqrt{5}}\left(\frac{\sqrt{5}+1}{2}\right)^n + \frac{1}{\sqrt{5}}\left(\frac{\sqrt{5}-1}{2}\right)^n$

不可以，精度

4) 有没有更快求解。（拓展）

$$F_{n+1} = F_n + F_{n-1} \iff \begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} F_1 \\ F_0 \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n}_{\text{只要求解}} \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$O(\log n)$

#2. 汉诺塔

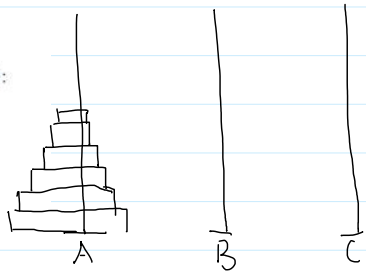
汉诺塔（港台：**河内塔**）（Tower of Hanoi）是根据一个传说形成的数学问题：

有三根杆子A，B，C。A杆上有 N 个 (N>1) 穿孔圆盘，盘的尺寸由下到上依次变小。要求按下列规则将所有圆盘移至 C 杆：

1. 每次只能移动一个圆盘；
2. 大盘不能叠在小盘上面。

提示：可将圆盘临时置于 B 杆，也可将从 A 杆移出的圆盘重新移回 A 杆，但都必须遵循上述两条规则。

问：如何移？最少要移动多少次？



n=1：　A → C　　1

n=2：　A → B　　3

n=1: A→C 1

n=2: A→B 3
A→C
B→C

n=3: A→C 7
A→B
C→B
A→C
B→A
B→C
A→C

Base Case: n=1的时候会移动吗?

假设已经会移动n-1个盘子了,如何移动n个盘子?

递推公式:
① 先将 n-1 盘子从起始杆移动到中间杆子上.
② 将最大的盘子从起始杆移动最终杆上
③ 将n-1个盘子从中间杆移动到最终杆上.

移动的次数 S(n)

$$S(n)=\begin{cases}1, & n=1\\ 2S(n-1)+1, & n\geq 2\end{cases}$$

$S(n)+1 = 2S(n-1)+1+1 = 2(S(n-1)+1) \Rightarrow S(n)=2^n-1$

怎么移?

```c
void hanoi(int n, char start, char middle, char target) {
    // base case
    if (n == 1) {
        printf("%c --> %c\n", start, target);
        return;
    }
    // 递推公式
    hanoi(n - 1, start, target, middle);
    // 把最大的盘子从start移动到target上
    printf("%c --> %c\n", start, target);
    hanoi(n - 1, middle, start, target);
}
```

总结: ① 什么可以考虑使用递归.
一个大问题可以分解为若干个小问题,且小问题的求解方式和大问题的求解一致. (递)
可以将这些小问题的解合并成大问题的解 (归)

② 使用递归需要注意的问题
边界条件,避免 stackoverflow 问题 ——→ stackoverflow.com 程序员的最爱.
重复计算

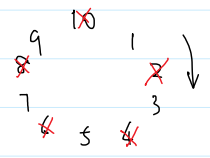
③ 如何写递归.
1) 边界条件
2) 递推公式.

约瑟夫环

人们站在一个等待被处决的圈子里。 计数从圆圈中的指定点开始，并沿指定方向围绕圆圈进行。 在跳过指定数量的人之后，处刑下一个人。 对剩下的人重复该过程，从下一个人开始，朝同一方向跳过相同数量的人，直到只剩下一个人，并被释放。

问题即，给定人数、起点、方向和要跳过的数字，选择初始圆圈中的位置以避免被处决。

n: 表示总人数, m: 每m个人杀一个人.



Joseph(10, 2) = 5

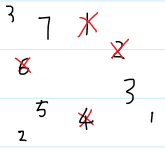
Joseph(n, m, k) = [Joseph(n, m) + (k-1)] % n

假定m=2. Joseph(n)

n为奇数 3 7

Joseph(n) = 2Joseph($\frac{n-1}{2}$) + 1

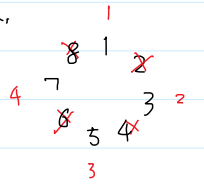
n为奇数



$Joseph(n) = 2\,Joseph\left(\frac{n-1}{2}\right) + 1$

$Joseph(7) = 7$

n为偶数,



$Joseph(n) = 2\,Joseph\left(\frac{n}{2}\right) - 1$

$Joseph(8) = 1$

Base case:

1) $n = 1$, $Joseph(1) = 1$

2) $n = 2$, $Joseph(2) = 1$

```
int joseph(int n) {
    if (n == 1 || n == 2) return 1;
    if (n & 0x1) {
        return 2 * joseph((n - 1) / 2) + 1;
    }
    return 2 * joseph(n / 2) - 1;
}
```

时间复杂度: $O(\log n)$

更一般的情况: Joseph(n, m). i.e. Joseph(7, 3) (作业).

	0	1	2	3	4	5	6
f(x):	3	4	5	6	0	1	
x:	0	1	2	3	4	5	

$f(x) = (x+3)\%7$

$Joseph(n,m) = (Joseph(n-1,m) + m)\%n$. (递推公式)

时间复杂度

2022年3月18日 14:38

$f(n), g(n): N^+ \to R^+$

$f = O(g)$，f 的增长速度不大于 g. 类比：$f \le g$.

当 n 足够大，$f(n) \le \underline{c} \cdot g(n) \Rightarrow f = O(g)$

c：某个常数

例子：$f_1(n) = n^2$，$f_2(n) = 2n + 20$. $f_2 = O(f_1)$

$$\frac{f_2}{f_1} = \frac{2n+20}{n^2} \le 22 \Leftrightarrow f_2 \le 22 f_1$$

$f_1 = O(f_2)$？ $\frac{f_1}{f_2} = \frac{n^2}{2n+20}$，同比 $f_1 \neq O(f_2)$

② $f = \Omega(g) \Leftrightarrow g = O(f)$，$f \ge g$，$f$ 的增长速度大于 g.

③ $f = \Theta(g) \Leftrightarrow f = O(g)$，$f = \Omega(g)$，$f = g$，$f$ 和 g 的增长速度相当的

诀窍：① 可以忽略系数，$14n^2 = \Theta(n^2)$

② 如果 $a > b$，则 $n^a > n^b$

③ 指数大于多项式：$3^n > n^5$

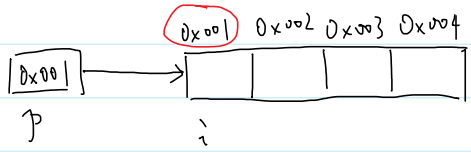
④ 多项式大于对数：$n > (\log n)^5$

指针（★★★★★）

2022年3月18日 16:17

指针基础、指针与数组的关系、指针的高级应用

1. 指针的基础



计算机最小的寻址单位：字节。

变量的地址：变量第一个字节的地址。

指针：简单来说，指针就是地址。

指针变量：存放地址的变量。有时候把指针变量称为指针。

(1) Q：指针变量只是存放了变量的首地址，那我们是怎么通过指针访问指针指向的对象。

声明指针时，需要指明基础类型，

```
int *p;
```

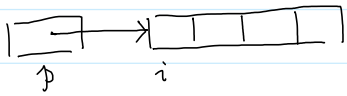
基础类型

→ 注意事项：{ 变量名为p，不是*p！
变量的类型为int*，而非int。

```
int *p, q;   // p为int*的指针，q是普通int类型。
int *p, *q;
```

(2) 两个基本操作：& 和 *。

取地址运算符：&　　`int i = 1;`　`int *p = &i;`



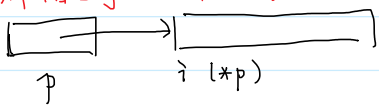
解引用运算符：*（通过指针访问指针指向的对象）

```
int i = 1;
int* p = &i;
*p = 2;
printf("i = %d\n", i);
```

选择Microsoft Visual Studio 调试控制台

```
i = 2
```

*p相当于i的别名，修改*p相当于修改i。



i：直接访问（访问一次内存）

*p：间接访问（访问两次内存）

(3) 野指针问题。

野指针：未初始化的指针或者指向未知区域的指针。

```
int *p;
int *q = 0x7F;
```

} 野指针

注意事项：对野指针进行解引用运算会导致未定义的行为。

```
int* p = 0x7F;
printf("*p = %d\n", *p);
```

已引发异常

引发了异常：读取访问权限冲突。
p 是 0x7F。

(4) 指针变量的赋值。

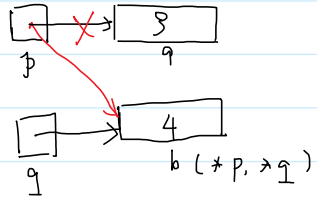
① 使用取地址运算符：`p = &i;`

② 通过为一个指针变量p赋值，p=q

注意事项：p=q和*p=*q的区别

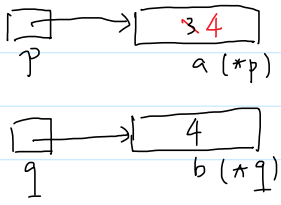
```c
int a = 3, b = 4;
int* p = &a;
int* q = &b;
p = q;
printf("a = %d, b = %d\n", a, b); // a = 3, b = 4
printf("*p = %d, *q = %d\n", *p, *q); // *p = 4, *q = 4;
```

```
Microsoft Visual Studio 调试控制台
a = 3, b = 4
*p = 4, *q = 4
```




```c
int a = 3, b = 4;
int* p = &a;
int* q = &b;
*p = *q;
printf("a = %d, b = %d\n", a, b);
printf("*p = %d, *q = %d\n", *p, *q);
```

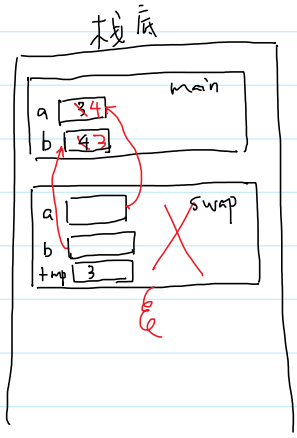
```
Microsoft Visual Studio 调试控制台
a = 4, b = 4
*p = 4, *q = 4
```



(5) 指针作为参数

```c
int main(void) {
    int a = 3, b = 4;
    printf("before: a = %d, b = %d\n", a, b);
    swap(&a, &b);
    printf("after: a = %d, b = %d\n", a, b);

    return 0;
}

void swap(int* a, int* b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

```
选择Microsoft Visual Studio 调试控制台
before: a = 3, b = 4
after: a = 4, b = 3
```



指针的好处：值传递是不够改变实参的值，而通过传递实参的地址，我们可以通过指针改变实参的值.

练习：找数组中最大元素和最小元素.

```c
void find_min_max(int arr[], int n, int* min, int* max) {
    *min = arr[0];
    *max = arr[0];
    for (int i = 1; i < n; i++) {
        if (arr[i] > *max) {
            *max = arr[i];
        } else if (arr[i] < *min) {
            *min = arr[i];
        }
    }
}
```

指针和数组 (☆☆☆☆☆)

2022年3月18日 17:32

指针和数组就像一对孪生兄弟。

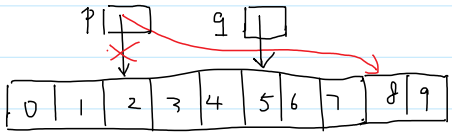
#1. 指针的算术运算

当指针指向数组元素时，可以通过指针的算术运算访问数组的其它元素。

① 指针加上一个整数

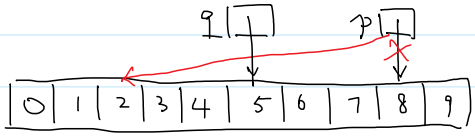
② 指针减去一个整数

③ 两个指针相减（指向同一个数组里面的元素）




```
int arr[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int* p = &arr[2];
int* q = p + 3;
p += 6;
printf("*p = %d, *q = %d\n", *p, *q);
```

```
Microsoft Visual Studio 调试控制台
*p = 8, *q = 5
```




```
int arr[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int* p = &arr[8];
int* q = p - 3;
p -= 6;
printf("*p = %d, *q = %d\n", *p, *q);
```

```
Microsoft Visual Studio 调试控制台
*p = 2, *q = 5
```

指针的算术运算是以元素大小为单位的，而不是以字节为单位。

```
int arr[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int* p = &arr[2];
int* q = &arr[5];
printf("p - q = %d\n", p - q);
printf("q - p = %d\n", q - p);
```

```
Microsoft Visual Studio 调试控制台
p - q = -3
q - p = 3
```

指针的比较（两个指针指向同一数组的元素）

$p-q>0 \Leftrightarrow p>q$

$p-q=0 \Leftrightarrow p=q$

$p-q<0 \Leftrightarrow p<q$

#2. 用指针处理数组

```
int arr[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int sum = 0;
for (int* p = &arr[0]; p < &arr[10]; p++) {
    sum += *p;
}
printf("sum = %d\n", sum);
```

&arr[10]是会计算 arr[10]的地址，不会访问 arr[10]，因此不会发生数组越界。

```
Microsoft Visual Studio 调试控制台
sum = 45
```

(1) * 和 ++ 的组合

*p++, *(p++)	表达式的值为 *p，副作用 p 自增。
(*p)++	表达式的值为 *p，副作用 *p 自增

*p++, *(p++)	表达式的值为*p，副作用p自增
(*p)++	表达式的值为*p，副作用*p自增
*++p, *(++p)	表达式的值为*(p+1)，副作用p自增
++*p, ++(*p)	表达式的值为*p+1，副作用*p自增

*和--也有类似的组合，不再赘述。

```c
int arr[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int sum = 0;
int* p = &arr[0];
while (p < &arr[10]) {
    sum += *p++;
}
printf("sum = %d\n", sum);
```