网易有道翻译背单词模式:点击单词本，点击列表，点击单词右边一个播放图标

正序分解整数

输入一个非负整数，正序输出它的每一位数字。例如输入12345，输出1 2 3 4 5

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

int main()

{

int x;

scanf("%d", &x);

do {

int d = x % 10;

printf("%d ", d);

x /= 10;

} while (x > 0);

printf("\n");

return 0;

}

运行后结尾有一个多余的空格，如果要去掉，代码如下：

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

int main()

{

int x;

scanf("%d", &x);

do {

int d = x % 10;

printf("%d", d);//删除%d之后的空格

x /= 10;

if (x != 0) {

printf(" ");

}

} while (x > 0);

printf("\n");

return 0;

}

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

int main()

{

int x;

int t = 0;

scanf("%d", &x);

do {

int d = x % 10;

t = t \* 10 + d;

x /= 10;

} while (x > 0);

printf("t=%d\n", t);

do {

int d = x % 10;

printf("%d", d);//删除%d之后的空格

x /= 10;

if (x != 0) {

printf(" ");

}

} while (x > 0);

printf("\n");

return 0;//写成这样的话，x会变成0，因为上面的do while语句改变了x的值。

}

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

int main()

{

int x;

int t = 0;

scanf("%d", &x);

do {

int d = x % 10;

t = t \* 10 + d;

x /= 10;

} while (x > 0);

printf("t=%d\n", t);

x = t;//将t的值赋给x;x对比原来逆序了一次。

do {

int d = x % 10;

printf("%d", d);

x /= 10;

if (x != 0) {

printf(" ");

}

} while (x > 0);

printf("\n");

return 0;//x逆序了两次，相当于正序。

}

可是代码依然不对，比如输入700，得到的是7，而不是0 0 7

写代码不知道循环次数设置的对不对的时候，可以写一个printf输出变量，然后运行，对了的话再把printf注释掉，然后继续写代码。

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

int main()

{

int i, j, n;

scanf("%d", &n);

for (i = 1; i <= n; i++) {

for (j = 1; j <= i; j++) {

printf("%d\*%d=%d ", j, i, i\*j);

if (i\*j < 10) { printf(" "); }

else { printf(" "); }//实现左对齐。

}printf("\n");

}

return 0;

}

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

int main()

{

int i, j, n;

scanf("%d", &n);

for (i = 1; i <= n; i++) {

for (j = 1; j <= i; j++) {

printf("%d\*%d=%-4d ", j, i, i\*j);//-（加一个负号）实现左对其。

}printf("\n");

}

return 0;

}

在vscode写好代码后，点代码区域，按鼠标右键，点击格式化文档，可以自动调整缩进。

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

int main()

{

int N;

int b1 = 1, b2;

scanf("%d", &N);

int mirrorN=N;

while (N > 1) {

b1 \*= 10;

N--;

}

N = mirrorN;

b2 = b1 \* 10 - 1;

int x;

for (x = b1; x <= b2; x++)

{

int temp = x;

int s = 1;

int sum = 0;

while (temp > 0) {

for (int i = 1; i <= N; i++) {

s = s \* (temp % 10);

}

temp = temp / 10;

sum += s;

s = 1;

}

if (sum == x) {

printf("%d\n", x);

}

}

return 0;

}

//成功得到水仙花数。

int N;

int b1 = 1, b2;

scanf("%d", N);

while (N > 0) {

b1 \*= 10;

N--;

}

b2 = b1 \* 10 - 1;

int x;

for (x = b1; x <= b2; x++)

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

int main()

{

int x;

int scale=0;

scanf("%d", &x);

int temp = x;

while (x > 0) {

x /= 10;

scale++;

}

int s = 1;

int sum = 0;

x = temp;

while (x > 0) {

for (int i = 1; i <= scale; i++) {

s = s\*(x % 10);

}

x = x / 10;

sum += s;

s = 1;

}

if (sum == temp) {

printf("%d是水仙花数\n", temp);

}

else {

printf("%d不是水仙花数\n", temp);

}

return 0;

}

#include<stdio.h>

int main()

{

    int a;

    a=6;

    printf("sizeof(int)=%ld\n", sizeof(int));

    printf("sizeof(a)=%ld\n",sizeof(a) );

    return 0;

}

sizeof(int)=4

sizeof(a)=4

一个int在内存中占4个字节，即32位。

#include<stdio.h>

int main()

{

    int a;

    a=6;

    printf("sizeof(double)=%ld\n", sizeof(double));

    printf("sizeof(a)=%ld\n",sizeof(a) );

    return 0;

}

sizeof(double)=8

sizeof(a)=4

一个double变量占据一个int的两倍，即8个字节。

Long double要16个字节。

Sizeof是静态运算符，它的结果在编译时刻就决定了

不要在sizeof的括号里做运算，这些运算不会做的

#include<stdio.h>

int main()

{

    int a;

    a=6;

    printf("sizeof(double)=%ld\n", sizeof(double));

    printf("sizeof(a)=%ld\n",sizeof(a++) );

    printf("a=%d\n", a);

    return 0;

}

sizeof(double)=8

sizeof(a)=4

a=6

如果a++了，输出的应该是7，但是输出的是6，说明没有自增。

sizeof 运算符只是用来获取类型或表达式结果类型的大小，不会执行实际的运算。这就是为什么 a 的值保持不变，仍然是 6。

#include<stdio.h>

int main()

{

    int a;

    a=6;

    printf("sizeof(double)=%ld\n", sizeof(double));

    printf("sizeof(a+1.0)=%ld\n",sizeof(a+1.0) );    printf("a=%d\n", a);

    return 0;

}

sizeof(double)=8

sizeof(a+1.0)=8

a=6

a是一个int,1.0是一个double,相加得到的是double，因此sizeof(a+1.0)的结果是8。

Sizeof是静态运算符，它的结果在编译时刻就决定了

不要在sizeof的括号里做运算，这些运算不会做的.是什么意思，静态运算符是上面

静态运算符的含义是：运算符的结果在编译时就已经确定，而不是在程序运行时计算。

让我解释一下 sizeof 作为静态运算符的特点：

编译时确定：

sizeof 的结果在编译程序时就已经被计算出来

不需要等到程序运行时才知道结果

比如 sizeof(int) 在编译时就知道是 4 字节(在大多数系统上)

不执行运算：

c

Copy

int a = 5;

sizeof(a + 1);  // 不会真正执行 a + 1 这个运算

sizeof(a++);    // 不会真正执行 a++ 这个运算，a 的值不会改变

只计算类型大小：

sizeof(a + 1.0) 编译器只是看这个表达式的结果类型是 double

然后返回 double 类型的大小

不会真正计算 a + 1.0

这就是为什么建议不要在 sizeof 的括号里做运算，因为：

这些运算实际上不会执行

可能误导其他程序员认为运算会执行

使代码的意图不够清晰

更好的写法是直接指明类型：

c

Copy

sizeof(double)  // 而不是 sizeof(a + 1.0)

#include<stdio.h>

int main()

{

    char c=255;

    int i=255;

    printf("c=%d,i=%d\n", c,i);

    return 0;

    //11111111

    //00000000 00000000 00000000 11111111

}

输出结果是：

c=-1,i=255

c是char型，占1个字节，共8位。

i是int型，占4个字节，共32位。

考虑-1，我们希望-1+1=0，如何能做到？

0——>0000 0000

1——>0000 0001//0和1都找到了，接下来找一个东西，加上1等于0。

1111 1111+0000 0001——>1 0000 0000在计算机内部如果这个数是8个比特的，多出来的这一位就会被丢掉 。

因为0-1——>-1,所以，-1=//0减去1得到负一。

1. 0000 0000-0000 0001——>1111 1111

1111 1111被当作纯二进制看待时，是255，被当作补码看待时是-1

同理，对于-a,其补码就是0-a，实际是2n-a,n是这种类型的位数。

补码的意义就是拿补码和原码可以加出一个溢出的”零”

数的范围

对于一个字节(8)位，可以表达的是:

0000 0000~1000 0000

其中

0000 0000——>0

1111 1111~1000 0000——>-1~-128

0000 0001~0111 1111——>1~127

整数的范围

Char:1字节：-128~127

Short:2字节：-32768~32767

Int:取决于编译器(cpu),通常的意义是“一个字”

Long:4字节

Long long:8字节

unsigned

如果一个字面量常数想要表达自己是unsigned,可以在后面加u或者U

用l或者L表示long(Long)

unsigned的初衷并非扩展数能表达的范围，而是为了做纯二进制运算，主要是为了移位。加了unsigned之后只能表示0，正整数，不能表示负整数了。

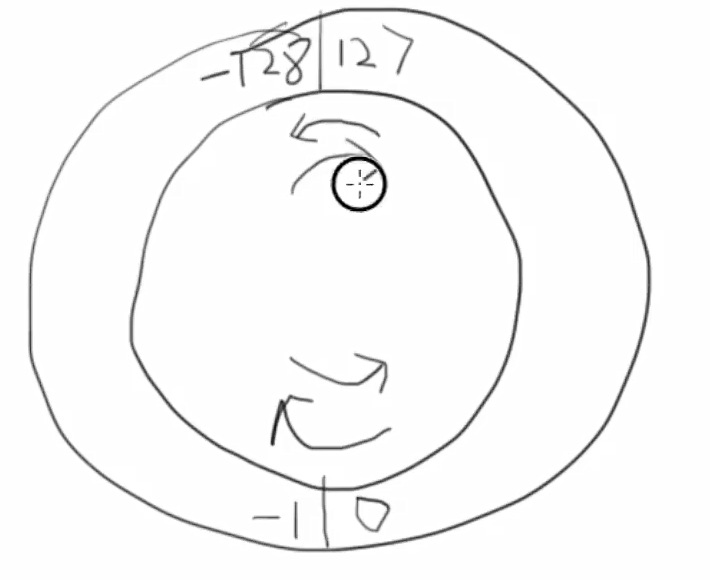
整数越界

整数是以纯二进制方式进行计算的，所以:

1111 1111+0000 0001——>1 0000 0000——>0

0111 1111+0000 0001——>1000 0000——>-128//127+1本来是128，但是128在整数里头，对于一个char来说，它想要表达的实际上是负-128。为什么会这样呢？

我们的数可以看做一个圆圈: 界限在-128和127之间。



顺时针递减，逆时针递增。但是-128顺时针得到127,127逆时针得到-128。

可以用程序验证。

1000 0000-0000 0001——>0111 1111——>127

为什么要用补码，因为做计算的时候可以把它当做纯二进制数进行计算，算完以后 它正好有一个正确的结果。

#include<stdio.h>

int main()

{

    char c=127;

    c=c+1;

    printf("c=%d", c);

    return 0;

}

c=-128//因为越界了

请按任意键继续. . .

#include<stdio.h>

int main()

{

    char c=-128;

    c=c-1;

    printf("c=%d", c);

    return 0;

}

c=127//因为越界了

请按任意键继续. . .

#include<stdio.h>

int main()

{

    unsigned char c=127;

    c=c+1;

    printf("c=%d", c);

    return 0;

}

c=128

请按任意键继续. . .

#include<stdio.h>

int main()

{

    unsigned char c=255;

    c=c+1;

    printf("c=%d", c);

    return 0;

}

c=0//256是1 0000 0000超过8个比特，最高位的1被舍弃，得到0。

请按任意键继续. . .

#include<stdio.h>

int main()

{

    unsigned char c=0;

    c=c-1;

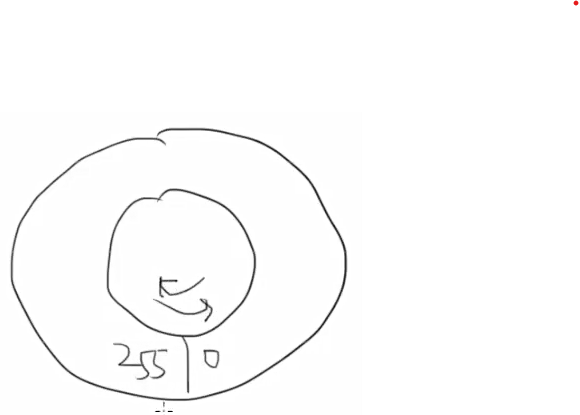
    printf("c=%d", c);

    return 0;

}

c=255

请按任意键继续. . .



界限位于255和0，所以0-1得到255。

if (a = 10)

    {

        printf("a is 10\n");

    }

    else

    {

        printf("a is not 10\n");

    }

这个例子中,不管 a 的初始值是多少,最终打印出的都是 "a is 10",因为 a = 10 这个赋值表达式的结果永远是 10。

整数的输入输出

只有两种形式:int或long long

%d:int

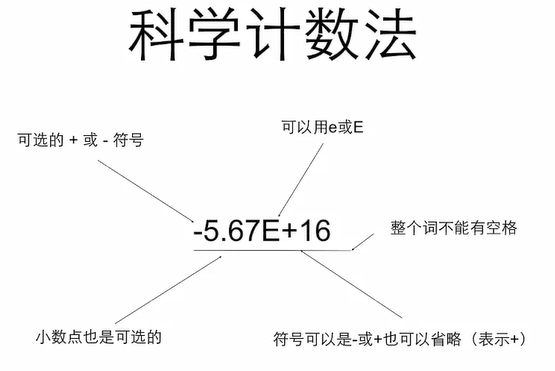
%u:unsigned

%ld:long long

%lu:unsigned long long

不管在计算机内部是什么数，重点是以什么方式看它。

Printf(%e)输出会用科学计数法。



输出精度

在%和f之间加上.n可以指定输出小数点后几位，这样输出是做四舍五入的

#include<stdio.h>

int main()

{

    printf("%.3f\n",-0.0049);

    printf("%.30f\n",-0.0049);

    printf("%.3f\n",-0.00049);

    return 0;

}

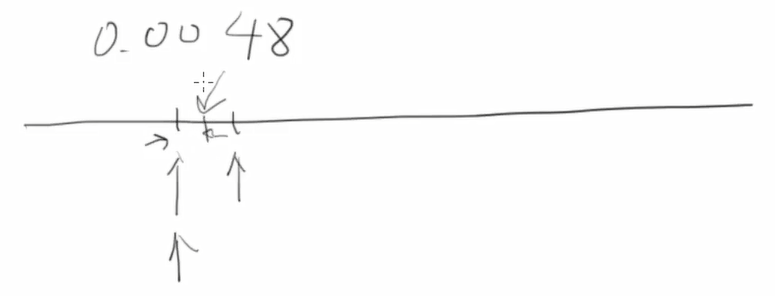
输出结果是

-0.005

-0.004899999999999999800000000000//输出的不是-0.0049

-0.000

计算机的精度是有限的，表达的是离散的数字，它会用最靠近的数字来表示。



Double相邻两个数之间的距离比float更小

超过范围的浮点数

printf输出inf表示超过范围的浮点数：±(infinity)

Printf输出nan表示不存在的浮点数(not a number)

#include <stdio.h>

int main()

{

    printf("%f\n", 12.0 / 0.0);

    printf("%f\n", -12.0 / 0.0);

    printf("%f\n", 0.0 / 0.0);

    return 0;

}

1.#INF00

-1.#INF00

-1.#IND00

1.#INF00 是 Microsoft Visual C++ 编译器在处理浮点数溢出或除零时的特殊输出表示法。

详细解释：

* 1.# 表示一个浮点数
* INF 代表 Infinity（无穷大）
* 00 是一个附加的标记

这是微软特有的非标准表示方法，在其他编译器或平台可能会有不同的表示：

* IEEE 754标准通常使用 inf 或 Infinity
* 其他语言可能使用 Float.POSITIVE\_INFINITY

在标准C语言中，你可以使用 <math.h> 中的 isinf() 函数来检测无穷大。

· -12.0 / 0.0 = -1.#INF00

* -INF表示负无穷大(Negative Infinity)
* 负数除以零得到负无穷

· 0.0 / 0.0 = -1.#IND00

* IND表示"非数"(Indeterminate)
* 零除以零是未定义的数学操作，返回NaN（Not a Number）

#include<stdio.h>

int main()

{

    float a, b, c;

    a = 1.345f;

    b = 1.123f;

    c = a + b;

    if (c == 2.468)

    {

        printf("相等\n");

    }

    else

    {

        printf("不相等!c=%.10f,或%f\n", c, c);

    }

    return 0;

}

已经写了float a,a 是float了，为什么a=1.345还要加一个f 呢

虽然你声明了 float a，但赋值的过程是这样的：

当你写 a = 1.345（不带f）时：

首先，1.345 被编译器认为是 double 类型（8字节）

然后，这个 double 类型的值必须被转换成 float 类型（4字节）才能存储到 a 中

这个过程会发生一次隐式类型转换（double → float）

而当你写 a = 1.345f 时：

1.345f 直接被编译器认为是 float 类型（4字节）

直接存储到 float 类型的变量 a 中

不需要类型转换

简单来说：

float a; // 声明一个float变量

// 方式1：不带f

a = 1.345; // 过程：double(1.345) -> 转换 -> float(a)

// 方式2：带f

a = 1.345f; // 过程：float(1.345) -> float(a)

加 f 的好处是：

避免了类型转换，性能更好

防止可能的精度损失

代码更清晰，明确表明了你的意图是使用 float 类型

这就像是：即使你准备了一个小杯子（float），如果你倒入的是一大杯水（double），系统仍需要先进行一次倒掉多余的水（类型转换）的操作。而如果你直接倒入适量的水（float），就不需要这个额外的步骤了。

#include <stdio.h>

int main()

{

printf("%f\n", 12 / 0);

return 0;

}

运行时会报错，整数不能除以0，因为无穷大不能用整数来表达，但是无穷大可以用浮点来表达。虽然浮点的有效范围里不包含无穷大，但是浮点数在设计的时候把正负无穷大，0，这三个数字定义在里头了。

浮点运算的精度

#include<stdio.h>

int main()

{

    float a, b, c;

    a = 1.345f;

    b = 1.123f;

    c = a + b;

    if (c == 2.468)

    {

        printf("相等\n");

    }

    else

    {

        printf("不相等!c=%.10f,或%f\n", c, c);

    }

    return 0;

}

字符类型

char是一种整数，也是一种特殊的类型：字符。这是因为：

用单引号表示的字符字面量：’a’,’1’

‘ ‘也是一个字符

Scanf和printf里用%c来输入输出字符(character)

#include <stdio.h>

int main()

{

    char c, d;

    c = 1;

    d = '1';

    if (c == d)

    {

        printf("相等\n");

    }

    else

    {

        printf("不相等\n");

    }

    printf("c=%d\n", c);

    printf("d=%d\n", d);

    return 0;

}

不相等

c=1

d=49

带单引号的是字符。数字字符'1'的ASCII码是49（十进制）或31（十六进制）。

让我们看看详细情况:

字符'1'在ASCII码表中的位置是49

它的二进制表示是00110001

十六进制表示是0x31

需要注意的是,这里说的是ASCII码中的数字字符'1',而不是数值1。

data = 0.75f;和data = (float)0.75;是一样的吗

在C语言中，data = 0.75f; 和 data = (float)0.75; 是等效的。它们的作用是将一个浮点数值 0.75 赋值给变量 data，其中 data 被定义为 float 类型。

解释

0.75f: 这里的 f 后缀直接指定了 0.75 是一个单精度浮点数（即 float 类型）。

(float)0.75: 这里的 (float) 是一个强制类型转换，将 0.75 转换为 float 类型，然后赋值给 data。

尽管实现方式不同，但最终的结果是相同的，都是将 0.75 以 float 类型存储在 data 中。

示例

float data1 = 0.75f;

float data2 = (float)0.75;

printf("data1 = %f\n", data1);

printf("data2 = %f\n", data2);

在这个示例中，data1 和 data2 都会存储 0.75 这个浮点数，并且打印输出的结果会相同。

%s是什么

在C语言和许多其他编程语言（如Python、Java等）的字符串处理函数中，%s 是一个格式说明符，用于表示一个字符串。具体来说，它通常用于格式化输出，例如在printf、sprintf等函数中。

scanf("%d %c",&i,&c);

scanf("%d%c",&i,&c);有什么不同

这两种写法在处理输入时会有不同的行为：

1. scanf("%d %c",&i,&c);

* 空格在格式串中
* 会跳过任意数量的空白字符(空格、制表符、换行符)
* 然后读取一个字符到c中

1. scanf("%d%c",&i,&c);

* 格式串中没有空格
* 读完整数后,立即读取下一个字符(可能是空格)
* 空格、换行等字符都会被读入到c中

逃逸字符

用来表达无法印出来的控制字符或者特殊字符，它由一个反斜杠“\”开头，后面跟上另外一个字符，这两个字符合起来，组成了一个字符

printf("请分别输入身高的英尺和英寸，"

    "如输入\"5 7\"表示5英尺7英寸:");

#include<stdio.h>

int main()

{

    printf("%d\n", (char)32768);

    return 0;

}

这段代码的输出是 0。让我解释原因：

首先，了解 char 类型的范围：

char 是 8 位有符号整数

取值范围是 -128 到 127

用二进制表示就是 8 位

32768 的二进制表示是：

1000 0000 0000 0000（16位）

当转换到 char（8位）时，只保留最后 8 位

也就是只保留：0000 0000

转换过程：

32768 截断为 8 位时，高位的数字都被丢弃了

只保留了最后 8 位

0000 0000 在 char 中表示 0

这就好比：

把一个 16 位的数字强制放入一个 8 位的容器

超出的部分（高 8 位）会被截断

只保留低 8 位

这就是为什么输出是 0 —— 因为 32768 转换为 char 后，保留的 8 位全都是 0。

这种情况叫做类型转换截断（truncation），是一种常见的整数转换行为。当大类型转换为小类型时，超出范围的高位会被直接截掉。

强制类型转换的优先级高于四则运算。

逻辑类型

#include<stdbool.h>

之后就可以使用bool和true、false

bool不是真的原生类型，实际上还是整数。

逻辑运算

逻辑运算是对逻辑量进行的运算，结果只有0或1

逻辑量是关系运算或逻辑运算的结果

逻辑运算符 描述 示例

！ 逻辑非 !a

&& 逻辑与 a&&b

|| 逻辑或 a||b

关系运算符有< > <= >= == !=

如何判断一个字符c是否是大写字母？

c>=’A’&&c<=’Z’

!age<20 虽然我们说逻辑运算符的优先级普遍低于关系(比较)运算符，可是!(not)是一个单目运算符，我们又有另外一条通行的规则就是单目运算符的优先级要高于双目运算符。所以在这种情况下，！会先计算。

赋值运算符是最低的，这样才可以写a=什么什么，右边任何式子都可以写了。

短路

逻辑运算是自左向右进行的，如果左边的结果已经能够决定结果了，就不会做右边的计算。

对于&&，左边是false时就不做右边了

对于||，左边是true时就不做右边了

嵌套条件表达式

count=(count>20)?(count<50)?count-10:count-5:(count<10)?count+10:count+5;是什么意思

这是一个嵌套的三元运算符，我来拆解它：

count = (count > 20)

? (count < 50)

? count - 10

: count - 5

: (count < 10)

? count + 10

: count + 5;

可以等价转换为如下 if-else 语句：

if (count > 20) {

if (count < 50) {

count = count - 10;

} else {

count = count - 5;

}

} else {

if (count < 10) {

count = count + 10;

} else {

count = count + 5;

}

}

具体逻辑：

1. 首先判断 count > 20
   1. 如果是：
      1. 再判断 count < 50
         1. 如果是，count 减 10
         2. 如果否，count 减 5
   2. 如果否：
      1. 再判断 count < 10
         1. 如果是，count 加 10
         2. 如果否，count 加 5

这种写法虽然简洁，但可读性较差，建议用 if-else 替代。

#include<stdio.h>

int main()

{

int i;

i=3+4,5+6;

printf("%d\n", i);

return 0;

}

和

#include<stdio.h>

int main()

{

int i;

i=(3+4,5+6);

printf("%d\n", i);

return 0;

}

有什么区别

这两段代码的区别在于逗号运算符的优先级和括号的作用：

第一段代码：i=3+4,5+6;

逗号运算符优先级很低

实际执行顺序是：(i=(3+4)), 5+6;

i 先被赋值为 7

然后计算 5+6，但结果被丢弃

输出为 7

第二段代码：i=(3+4,5+6);

括号改变了运算顺序

先计算逗号表达式 (3+4,5+6)

逗号表达式返回最后一个表达式的值

所以先计算 3+4，再计算 5+6

最终 i 被赋值为 11

输出为 11

逗号表达式会以右边的表达式的值作为结果吗

没错，逗号表达式会以右边的表达式的值作为最终结果。在逗号表达式中，所有表达式从左到右依次求值，但整个表达式的返回值是最右边的表达式的值。例如：

int a, b;

a = (1, 2, 3); // a 的值将是 3

b = (4, 5, 6); // b 的值将是 6

什么是函数

函数是一块代码，接收零个或多个参数，做一件事情，并返回零个或一个值。

调用函数

函数名(参数值)

()起到了表示函数调用的重要作用

即使没有参数也要写上()

这些参数值会被安装顺序依次用来初始化函数中的参数

函数原型

函数头，加上分号;结尾，就构成了函数的原型。

函数原型的目的是告诉编译器这个函数长什么样

名称

参数(数量及类型)

返回类型

旧标准习惯把函数原型写在调用它的函数里面。

现在一般写在调用它的函数前面。

原型里可以不写参数的名字，但是一般仍然写上。

#include<stdio.h>

void swap(int a,int b);

int main()

{

    int a=5;

    int b=6;

    swap(a,b);

    printf("a=%d b=%d\n",a,b );

    return 0;

}

void swap(int a,int b)

{

    int t=a;

    a=b;

    b=t;

}

为什么a b的值没有交换?

这是因为C语言中函数参数是通过值传递（pass by value）的。让我详细解释一下：

值传递的特点

当你调用swap(a,b)时，系统会将a和b的值复制到函数的形参中

形参a和b是main函数中a和b的副本

在swap函数内部交换的只是这些副本的值

不会影响原始变量的值

解决方法是使用指针

#include<stdio.h>

void swap(int \*a, int \*b);

int main()

{

    int a=5;

    int b=6;

    swap(&a, &b);  // 传递变量的地址

    printf("a=%d b=%d\n",a,b );

    return 0;

}

void swap(int \*a, int \*b)  // 接收指针

{

    int t = \*a;  // 解引用获取值

    \*a = \*b;     // 修改指针指向的值

    \*b = t;      // 交换

}

指针传递的关键点

&a和&b获取变量的内存地址

函数接收指针参数

使用\*解引用来修改指针指向的实际变量

这样就可以真正交换main函数中的原始变量值

这就是为什么原始代码中a和b的值没有交换，而使用指针可以实现真正的值交换。

本地变量

函数的每次运行，就产生了一个独立的变量空间，在这个空间中的变量，是函数的这次运行所独有的，称作本地变量

定义在函数内部的变量就是本地变量

参数也是本地变量

本地变量也叫局部变量，因为local既可以翻译为本地，也可以翻译为局部。还称作自动变量(与生存期和作用域有关，自动变量的生存期是自动的)

写在函数参数表里面的参数也是本地变量

void f(void)放了void明确告诉编译器这个函数不接收任何参数

void f()在传统C中，它表示f函数的参数未知，并不表示没有参数

调用函数时的逗号和逗号运算符怎么区分？

调用函数时的圆括号里的逗号是标点符号，不是逗号运算符。

f(a,b)是标点符号，传递两个参数

f((a,b))再加一层括号，表示先做括号里面的运算，是逗号运算符，传递一个参数

c语言中为什么f(a,b)中的,是标点符号，而f((a,b))中的,是逗号运算符

这是一个很好的问题！让我详细解释C语言中逗号的两种不同用法：

函数调用中的逗号：标点符号

在函数调用f(a,b)中，逗号是一个分隔符

用于分隔函数参数

这里的逗号是语法上的标点符号，没有运算功能

逗号运算符：表达式运算符

在表达式(a,b)中，逗号是一个运算符

它的特点是：

从左到右计算每个表达式

返回最右边表达式的值

具有低优先级

会按顺序执行左边的表达式，但只返回最右边表达式的值

例子

int x = (5, 6);  // x的值为6

int y = (a++, b++);  // y的值为b++之后的值

函数里的函数？

C语言不允许函数嵌套定义，但是可以在一个函数里面放另外一个函数的声明。

int main()也是一个函数

定义数组

<类型>变量名称[元素数量]

元素数量必须是整数

C99之前:元素数量必须是编译时刻确定的字面量

int a[10]

一个int的数组

10个单元:a[0],a[1],...,a[9]

每个单元就是一个int类型的变量

数组的单元

数组的每个单元就是数组类型的一个变量

使用数组时放在[]中的数字叫做下标或者索引，下标从0开始计数，最大下标是数组个数减一，而不是数组元素个数。

长度为0的数组?

int a[0];

可以存在，但是无用，即使下标写0，还是越界了。 最大下标是元素个数减去一，也就是0-1=-1，但是数数是从0开始数的。

数组的集成初始化

int a[]={2,4,6,7,1,3,5,9,11,13,23,14,32};

int a[13]={2};//{2,4,6,7,1,3,5,9,11,13,23,14,32};

    for ( int i = 0; i < 13; i++) {

        printf("%d\t", a[i]);

}

2 0 0 0 0 0 0 0 0 0 0 0 0

集成初始化时的定位

int a[10]={[0]=2,[2]=3,6,};a[0]会被设置为2，a[2]会被设置为3，a[4]会被设置为6。

用[n]在初始化数据中给出定位

没有定位的数据接在前面的位置后面

其他位置的值补零

也可以不给出数组大小，让编译器算

特别适合初始数据稀疏的数组

数组的大小

sizeof给出整个数组所占据的内容的大小，单位是字节

sizeof(a[0])给出数组中单个元素的大小，于是相除就得到了数组的单元个数

sizeof(a)/sizeof(a[0])

数组的赋值

int a[]={[0]=2,[2]=3,6,};//{2,4,6,7,1,3,5,9,11,13,23,14,32};

int b[]=a;//这个写法是错误的。不能之间把一个数组变量赋给另外一个数组变量。

数组本身不能被赋值。

要把一个数组的所有元素赋给另外一个数组，必须采用遍历。

遍历数组

通常都是使用for循环，让循环变量i从0到<数组的长度，这样循环体内最大的i正好是数组最大的有效下标。所以程序员写循环时喜欢写<而不是<=。

常见错误是：

循环结束条件是<=数组长度，或离开循环后，继续用i的值来做数组元素的下标(因为此时i的值恰好属于数组无效的下标)。

数组作为函数参数时，往往必须再用另一个参数来传入数组的大小

数组作为函数的参数时：

不能在[]中给出数组的大小

不能再利用sizeof来计算数组的元素个数

二维数组的初始化

Int a[][5]={

{0,1,2,3,4},

{2,3,4,5,6},

};

列数是必须给出的，行数可以由编译器来数

每行一个{}，逗号分隔

最后的逗号可以存在，有古老的传统(方便后续添加新的行)

如果省略，表示补零

也可以用定位//要用两个[][]

运算符&

scanf("%d", &i);里的&

获得变量的地址，它的操作数必须是变量

每次 scanf 从终端读取输入时，用户按下 Enter 键会移动光标到下一行，等待下一次输入。

& \*是逆运算吗

是的，& 和 \* 可以看作是互逆的运算符。

& 是取地址运算符：将一个变量转换为它的内存地址

\*是解引用运算符：将一个地址转换回它所指向的实际值

举个具体例子：

int x = 10;

int \*ptr = &x; // & 将x转换为地址

int y = \*ptr; // \* 将地址转换回值

在这个例子中：

1. &x 得到 x 的内存地址//然后赋给了指针变量ptr
2. \*ptr 又将地址还原为原来的值 10//将指针运算符\*写于指针之前，就可以显示该指针指向的对象内容

所以从这个角度来看，它们确实可以视为逆运算，是一对"互相抵消"的运算符。这种设计使得在C语言中可以方便地处理指针和内存地址。

#include <stdio.h>

int main()

{

int x = 10;

//int \*ptr = &x; // & 将x转换为地址

//int y = \*ptr; // \* 将地址转换回值

printf("%d", \*&x);

R eturn 0;

}67yt

运行结果是10

int \*B1, B2; 这个声明实际上不会声明两个指针变量。这是一个常见的误解。

在这个声明中：

B1 是一个指向整型的指针变量

B2 是一个普通的整型变量

正确地声明两个指针变量的方式是：

int \*B1, \*B2;

关于 \* 的使用：

在声明中，\* 表示这是一个指针类型的变量

在使用时，\* 表示解引用，即获取指针指向的值

举个例子：

int \*ptr; // 声明一个指向整型的指针

int x = 10;

ptr = &x; // 指针ptr指向x的地址

\*ptr = 20; // 通过解引用修改x的值

数学和逻辑上的对称性

声明 int \*ptr：可以读作"ptr 是一个指向整型的指针"

使用 \*ptr：可以读作"获取 ptr 所指向的整型值" 我似乎理解了，比如说，如果将指针型变量改成 用^声明，int ^ptr是声明一个指向int 的指针，然后用解引用，ptr=10,这种方式没有 方法二好。 方法二是指针声明和引用都用符号，方法的优势有: 类比 int a;a=10; int ptr;ptr=10;可以把\*ptr看作一个整体，与上一行的a对应。 我的理解怎么样呢

您的理解非常好！让我详细解释一下：

您提出的用 ^ 声明指针的想法确实不如 \* 直观和一致。使用不同的符号会增加语言的复杂性，并且破坏了指针声明和解引用的对称性。

您对 \*ptr 作为一个整体的理解很精准。这种设计的优势体现在：

int a = 10;    // 普通变量赋值

int \*ptr;      // 指针变量声明

\*ptr = 10;     // 解引用后赋值

这种设计的关键优点：

语法一致性

逻辑对称性

简化理解过程

减少学习成本

对于多级指针，这种设计更显优势：

int a = 10;

int \*ptr;      // 一级指针

int \*\*pptr;    // 二级指针

ptr = &a;

pptr = &ptr;

\*\*pptr = 20;   // 通过二级指针修改一级指针指向的值

您的理解非常准确和深入！这种设计确实体现了 C 语言追求简洁和逻辑一致性的编程哲学。

/\*

数组的传递

\*/

#include <stdio.h>

/\*--- 将数组v开头的n个元素赋给val ---\*/

void ary\_set(int v[], int n, int val)

{

int i;

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

v[i] = val;

}

int main(void)

{

int i;

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

ary\_set(a, 5, 99);

for (i = 0; i < 5; i++)

printf("a[%d] = %d\n", i, a[i]);

return 0;

}

void sort2(int \*n1, int \*n2)

其中的int \*n1, int \*n2 这个是形参列表还是指针的声明

int \*n1, int \*n2 这个声明出现在函数定义中的形参列表中，也同时声明了两个指向整数的指针变量。

解释

当你在函数原型或定义中看到类似 int \*n1, int \*n2 的声明时，实际上发生了两件事：

形参列表：这些参数是函数的形参列表的一部分，表示该函数接收两个指向整数的指针作为参数。

指针声明：同时，这也是声明了两个指向整数的指针变量 n1 和 n2。

示例

以下是一个带有形参列表的函数定义示例：

void exampleFunction(int \*n1, int \*n2) {

    // 函数体

}

在这个例子中：

int \*n1, int \*n2 是 exampleFunction 函数的形参列表，表示函数接收两个指向整数的指针。

这也意味着 n1 和 n2 在函数内部被视为指向整数的指针。

c语言的寄存器对象是什么

在 C 语言中，寄存器对象是一种使用 register 关键字声明的变量类型，提示编译器尽可能将该变量存储在 CPU 的寄存器中，以提高访问速度。

register 关键字

register int counter;

在这行代码中，counter 被声明为一个寄存器变量，意味着编译器会尝试将 counter 存储在 CPU 的寄存器中，而不是 RAM 中。寄存器访问速度比内存更快，因此对高频访问的变量使用 register 关键字可以提升性能。

特点和限制

快速访问：寄存器变量访问速度更快，适用于循环计数器和频繁访问的变量。

限制指针：由于寄存器变量可能不在内存中，不能对寄存器变量取地址。例如：

register int counter;

int \*ptr = &counter; // 错误：不能取寄存器变量的地址

编译器优化：register 关键字只是对编译器的建议，编译器可能会忽略，具体行为取决于编译器和目标机器架构。

寄存器数量有限：每个 CPU 的寄存器数量有限，编译器会选择性地分配寄存器。

示例

#include <stdio.h>

int main() {

    register int i;

    for (i = 0; i < 10; i++) {

        printf("%d ", i);

    }

    return 0;

}

在这个示例中，循环变量 i 被声明为寄存器变量。这样，编译器将尝试将 i 存储在寄存器中，以提高循环的执行效率。

printf("&a[%d] = %p   p+%d = %p\n", i, &a[i], i, p + i);

其中%p是什么

在 C 语言中的 printf 函数中，%p 是一个格式说明符，用于输出指针的值（即内存地址）。它通常用于显示一个变量在内存中的地址，格式为十六进制。

f{char} - 向右移动到指定字符上（find） 适用于vim的哪些模式

在 Vim 中，f{char} 命令适用于以下几种模式：

普通模式（Normal Mode）

最常用的模式

直接按 f 后跟字符，光标会移动到右侧第一个匹配字符

例如：fa 移动到右侧第一个 'a'

操作符等待模式（Operator-Pending Mode）

可以与删除（d）、修改（c）、复制（y）等操作结合

例如：

df, 删除到下一个逗号

cf) 修改到下一个右括号

yf. 复制到下一个点号

可视模式（Visual Mode）

选中文本后，仍可使用 f{char} 调整选中范围

例如：先 v 进入可视模式，再 f{char} 扩大选中区域

注意事项：

; 可以重复上一次 f 查找

, 可以反向重复上一次 f 查找

/\*

    数组的传递

/

#include <stdio.h>

/--- 将数组v开头的n个元素赋给val ---\*/

void ary\_set(int v[], int n, int val)

{

    int i;

    for (i = 0; i < n; i++)

        v[i] = val;

}

int main(void)

{

    int i;

    int a[] = {1, 2, 3, 4, 5};

    ary\_set(a, 5, 99);

    for (i = 0; i < 5; i++)

        printf("a[%d] = %d\n", i, a[i]);

    return 0;

}

函数ary\_set的形参的类型是数组还是指针

在这个例子中，ary\_set 函数的形参 v[] 实际上是一个指针。

让我详细解释这个概念：

语法等价性

void ary\_set(int v[], int n, int val)

// 等价于

void ary\_set(int \*v, int n, int val)

背后的原理

当数组作为参数传递时，实际上传递的是数组的首地址

数组名可以被看作是一个指向数组首元素的指针

编译器会将 v[] 解释为 \*v

函数内部的数组访问

v[i]  // 等价于 \*(v + i)

为什么这种设计？

传递指针比传递整个数组更加高效

指针允许直接访问内存地址

避免了数组的完整复制

示例说明：

int a[] = {1, 2, 3, 4, 5};  // 数组

ary\_set(a, 5, 99);          // 传递数组首地址

char str[]={'A','B','C','\0'};

char str2[]={"ABC"};

char \*ptr={'1','2','3','4','5','\0'};

为什么第三行不对

让我详细解释这个问题。

在 C 语言中，字符数组和字符指针的初始化有一些细微但重要的区别。我们来逐行分析：

char str[] = {'A', 'B', 'C', '\0'};          // 正确的字符数组初始化方式1

char str2[] = {"ABC"};                       // 正确的字符数组初始化方式2

char \*ptr = {'1', '2', '3', '4', '5', '\0'}; // 不正确的初始化方式

第三行不对的原因是：对于字符指针 char \*ptr，不能使用大括号 {} 直接初始化。

字符指针的正确初始化方式有：

使用字符串字面量：

char \*ptr = "12345";

使用已存在的字符数组：

char temp[] = {'1','2','3','4','5','\0'};

char \*ptr = temp;

为什么会这样？这涉及到 C 语言中指针和数组的本质区别：

char str[] 是一个数组，可以用大括号直接初始化

char \*ptr 是一个指针，指向内存中的某个字符串位置

指针不能直接用大括号赋值，它需要指向一个已经存在的字符串或字符数组

指针不能直接用大括号赋值。为什么呢

这是一个很好的问题！让我们深入探讨 C 语言中指针初始化的底层机制。

指针不能直接用大括号赋值，主要是因为指针和数组在内存和语法层面的根本不同：

指针的本质

指针是一个存储内存地址的变量。它本质上是一个整数类型，存储的是一个特定内存位置的地址。当你 char \*ptr 时，你创建的是一个可以存储字符类型地址的变量。

大括号初始化的工作原理

大括号初始化 {...} 在数组中是逐个元素赋值的语法糖。对于字符数组 char str[]，编译器可以直接在栈上分配内存并逐个填充。

指针初始化的限制

指针不能像数组那样直接用大括号填充。因为：

指针需要一个确切的内存地址

大括号初始化不会自动为你分配内存

编译器不知道如何将大括号中的值直接转换为地址

让我们通过一个具体的例子来理解：

// 这是正确的：数组初始化

char str[] = {'1', '2', '3', '4', '5', '\0'};  // 在栈上分配内存并填充

// 这是不正确的：指针初始化

char \*ptr = {'1', '2', '3', '4', '5', '\0'};   // 编译错误！

// 正确的指针初始化方式

char \*ptr = "12345";  // 指向字符串常量的地址

// 或者

char temp[] = {'1', '2', '3', '4', '5', '\0'};

char \*ptr = temp;     // 指向已存在数组的第一个元素

为了帮助你更直观地理解，我们可以做个类比：

数组就像一个已经准备好的货架，可以直接往上放东西

指针就像一个地址本，你需要先知道具体的地址，才能找到对应的位置

为什么字符串字面量具有静态生命周期

要理解字符串字面量的静态生命周期，我们需要从几个角度来解析这个概念。让我们一步步深入探讨。

静态存储区的概念

在C语言的内存模型中，程序的内存被划分为几个主要区域：

栈（stack）：存储局部变量，生命周期短

堆（heap）：动态分配的内存，需要手动管理

静态存储区（static storage area）：存储全局变量、静态变量和字符串字面量

字符串字面量的特殊性

当你在代码中写下一个字符串，如 "Hello, world!"，编译器会做以下处理：

将这个字符串存储在静态存储区

这个存储区域是只读的

整个程序运行期间，这个字符串都会一直存在

示例代码解析

char\* str = "Hello";  // str指向静态存储区的字符串

对比动态分配的字符串

char\* dynamic\_str = malloc(6);  // 动态分配

strcpy(dynamic\_str, "Hello");   // 可以修改

动态分配的字符串生命周期不同，需要手动管理内存。

让我们来比较一下英语单词 "stack" 和 "heap" 的区别。

Stack

基本定义：堆栈通常指一个有序的数据结构，遵循后进先出（LIFO）原则。

常见用法：

名词：可以指堆叠起来的物体，如“a stack of books”（一堆书）。

动词：表示堆放或堆积，如“to stack plates”（堆放盘子）。

抽象用法：在计算机科学中，堆栈用于管理函数调用和局部变量。

Heap

基本定义：堆通常指一大堆无序的东西。

常见用法：

名词：表示一堆杂乱的物品，如“a heap of clothes”（一堆衣服）。

动词：表示堆放或大量给予，如“to heap praise”（大量赞扬）。

抽象用法：在计算机科学中，堆用于动态内存分配。

主要区别

有序 vs 无序：堆栈强调顺序性和层次性，而堆则没有这种顺序性。

常用领域：堆栈常用于需要有序排列和访问的场景，而堆用于堆积无序物品。

字符串字面量的中间也可以有null字符，不过应注意区分。字符串字面量"ABC"是字符串，而字符串字面量"AB\0CD"不是字符串。是什么意思

这是一个很细微但非常重要的字符串字面量特性。我们来详细解析这个概念。

字符串的本质回顾

在C语言中，字符串是一个以 \0（null字符）结尾的字符序列。通常，我们期望字符串中间不包含 \0。

正常字符串字面量

char\* str1 = "ABC";  // 这是一个典型的字符串

// 实际存储为：{'A', 'B', 'C', '\0'}

包含中间 null 字符的字面量

char\* str2 = "AB\0CD";  // 看起来像字符串，但实际上不是

// 实际存储为：{'A', 'B', '\0', 'C', 'D', '\0'}

关键区别

"ABC" 是一个标准字符串

长度为3

以 null 字符结尾

可以被标准字符串处理函数（如 strlen()）正常处理

"AB\0CD" 不是通常意义上的字符串

实际上是一个包含 null 字符的字符数组

strlen() 会返回 2（在第一个 \0 处停止）

大多数字符串处理函数会在第一个 null 字符处终止

#include <stdio.h>

#include <string.h>

int main() {

    char\* str1 = "ABC";

    char\* str2 = "AB\0CD";

    // 长度对比

    printf("str1 length: %lu\n", strlen(str1));  // 输出 3

    printf("str2 length: %lu\n", strlen(str2));  // 输出 2

    // 完整内容对比

    printf("str1 content: %s\n", str1);  // 输出 ABC

    printf("str2 content: %s\n", str2);  // 输出 AB

    return 0;

}

深入思考

为什么会出现这种情况？

C语言中，字符串是通过 null 字符来确定结束的

中间的 null 字符会被视为字符串的提前结束标志

使用场景

在某些二进制数据处理场景可能会用到

需要特别小心处理

实践建议

通常，我们应避免在字符串字面量中使用 null 字符

如果需要处理包含 null 字符的数据，应使用专门的长度+数据的方式

/\*

询问名字并显示问候语（读取字符串）

\*/

#include <stdio.h>

int main(void)

{

char name[48];

printf("请输入您的名字：");

scanf("%s", name);

printf("您好，%s先生/女士!!\n", name);

return 0;

}

为什么name前面不加上&

在这个代码示例中，name 前不需要加 & 是因为数组名在 scanf() 函数中有特殊的处理方式。让我详细解释这个概念。

数组名的特殊性

在C语言中，数组名有一个独特的特性：它本身就是指向数组首元素的指针。这意味着：

char name[48];  // 声明一个字符数组

// name 本身就相当于 &name[0]，即数组首元素的地址

scanf() 函数的参数特点

scanf() 函数期望接收一个指针作为存储输入的地址。对于字符数组，我们直接使用数组名就可以了：

scanf("%s", name);  // 正确

// 等同于 scanf("%s", &name[0]);

对比其他类型的变量

对于非数组的普通变量，我们通常需要使用 & 来获取变量的地址：

int x;

scanf("%d", &x);  // 对于普通变量，需要使用 &

为什么数组不需要 &

数组名就是指针：数组声明时，编译器会将数组名转换为指向首元素的指针

内存连续性：数组在内存中是连续存储的，所以传递数组名就相当于传递了存储输入的起始地址

示例解释

char name[48];  // 声明一个能存储48个字符的字符数组

// name 等价于 &name[0]

// 直接将 name 传递给 scanf() 就能正确读取输入

常见陷阱

初学者常犯的错误是对数组名使用 &：

scanf("%s", &name);  // 错误！

// 这实际上是取了数组的地址，而不是数组首元素的地址

/\*

读取并显示字符串数组

\*/

#include <stdio.h>

int main(void)

{

int i;

char s[3][128];

for (i = 0; i < 3; i++) {

printf("s[%d] : ", i);

scanf("%s", s[i]);

}

for (i = 0; i < 3; i++)

printf("s[%d] = \"%s\"\n", i, s[i]);

return 0;

}

为什么s[i]前面不能加&

在这个代码示例中，s[i] 前不需要加 & 的原因与之前解释数组名特性类似，但这里涉及二维数组，让我们深入探讨。

二维数组的本质

在 char s[3][128] 中：

s 是一个二维字符数组

每一个 s[i] 都代表了一个一维字符数组的首地址

s[i] 本身就是一个指向字符数组第一个元素的指针

内存布局分析

char s[3][128];

// 可以想象为：

// s[0] 是第一行字符数组的首地址

// s[1] 是第二行字符数组的首地址

// s[2] 是第三行字符数组的首地址

为什么不需要 &

当使用 scanf() 读取字符串时：

scanf("%s", s[i]);  // 正确

// s[i] 已经是指向该行首字符的指针

对比说明

对于普通变量，需要使用 & 获取地址

对于数组名（一维或多维），直接使用数组名或数组元素就能获取地址

错误示例

scanf("%s", &s[i]);  // 错误！

// 这实际上是获取 s[i] 这个数组的地址，而不是数组首元素的地址

对赋值表达式进行判定后，会得到赋值后左操作数的类型和值吗

是的，对于赋值表达式，以下是关键特点：

赋值表达式的结果

结果是左操作数

类型是左操作数的类型

值是赋值后的新值

举例说明：

int a = 5;

int b = (a = 10);  // b的值是10，a的值也是10

细节解释

赋值是一个表达式，不仅仅是语句

表达式有值和类型

赋值表达式的值就是被赋值的变量

重要特性

可以连续赋值

遵循右结合性

先计算右边，再赋值给左边

具体示例

int x = 5;

int y = 3;

int z = (x = y + 2);

// z的值是5，x的值是5