第10讲：操作符详解

目录

1. 操作符的分类
2. 二进制和进制转换
3. 原码、反码、补码
4. 移位操作符
5. 位操作符：&、|、^、~
6. 单目操作符
7. 逗号表达式
8. 下标访问[]、函数调用()
9. 结构成员访问操作符
10. 操作符的属性：优先级、结合性
11. 表达式求值

---

1. 操作符的分类

- 算术操作符：`+ 、- 、* 、/ 、%`
- 移位操作符：`<< >>`
- 位操作符：`& | ^`
- 赋值操作符：`= 、+= 、 -= 、 *= 、 /= 、%= 、<<= 、>>= 、&= 、|= 、^=`
- 单目操作符：`! 、++、--、&、*、+、-、~ 、sizeof、(类型)`
- 关系操作符：`> 、>= 、< 、<= 、 == 、 !=`
- 逻辑操作符：`&& 、||`
- 条件操作符：`? :`
- 逗号表达式：`,`
- 下标引用：`[]`
- 函数调用：`()`

- 结构成员访问：`.` 、`->`

上述的操作符，我们已经讲过算术操作符、赋值操作符、逻辑操作符、条件操作符和部分的单目操作符，今天继续介绍一部分，操作符中有一些操作符和二进制有关系，我们先铺垫一下二进制的和进制转换的知识。

2. 二进制和进制转换

其实我们经常能听到`2进制、8进制、10进制、16进制`这样的讲法，那是什么意思呢?

其实2进制、8进制、10进制、16进制是数值的不同表示形式而已。

比如：数值15的各种进制的表示形式：

代码块

```
15的2进制：1111
15的8进制：17
15的10进制：15
15的16进制：F

//16进制的数值之前写：0x
//8进制的数值之前写:0
```

我们重点介绍一下二进制：

首先我们还是得从10进制讲起，其实10进制是我们生活中经常使用的，我们已经形成了很多常识：

- 10进制中满10进1
- 10进制的数字每一位都是0~9的数字组成

其实二进制也是一样的

- 2进制中满2进1
- 2进制的数字每一位都是0~1的数字组成

那么`1101`就是二进制的数字了。

2.1 2进制转10进制

其实10进制的123表示的值是一百二十三，为什么是这个值呢？其实10进制的每一位是有**权重**的，10进制的数字从右向左是个位、十位、百位....，分别每一位的权重是$10^0, 10^1, 10^2 ...$

如下图：

	百位	十位	个位	
10进制的位	1	2	3	
权重	10^2	10^1	10^0	
权重值	100	10	1	
求值	1*100 +	2*10 +	3*1 =	123

10进制123每一位权重的理解

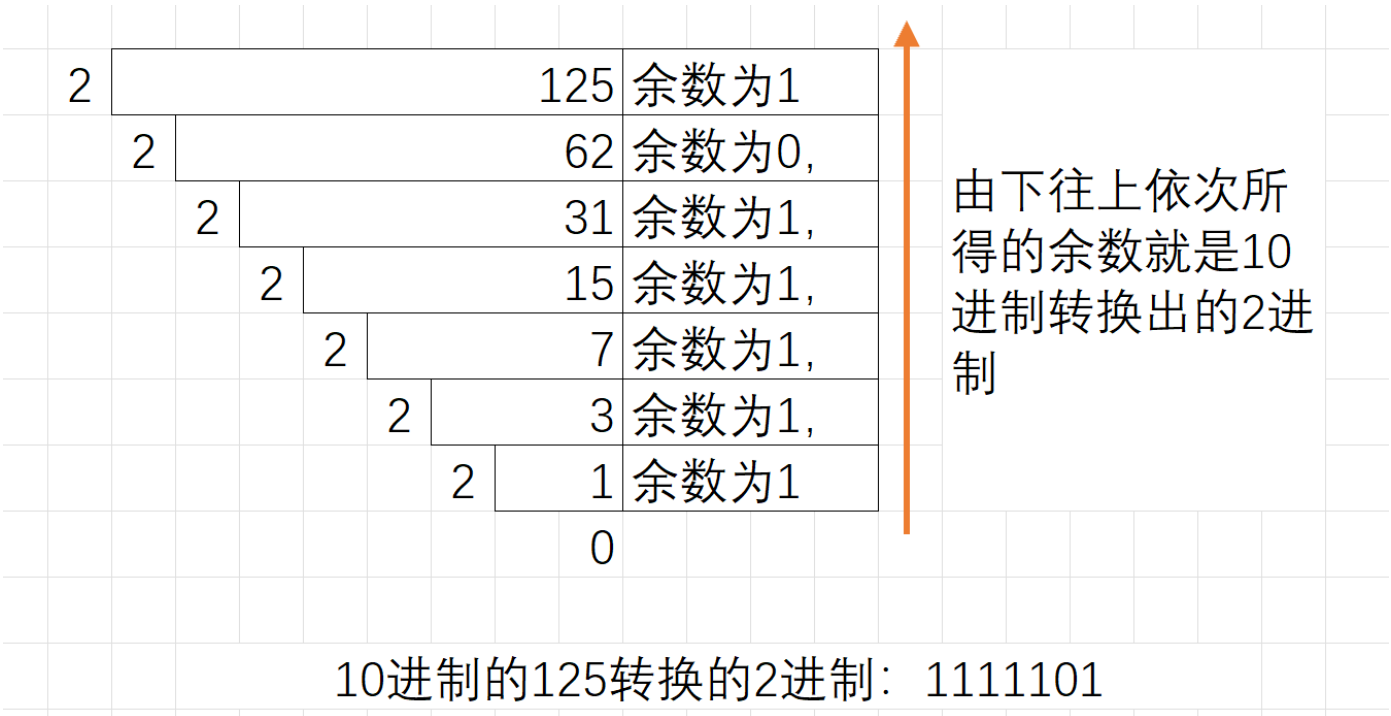
2进制和10进制是类似的，只不过2进制的每一位的权重，从右向左是: $2^0, 2^1, 2^2 \ldots$

如果是2进制的1101，该怎么理解呢?

2进制的位	1	1	0	1	
权重	2^3	2^2	2^1	2^0	
权重值	8	4	2	1	
求值	1*8 +	1*4 +	0*2 +	1*1 =	13

2进制1101每一位权重的理解

2.1.1 10进制转2进制数字



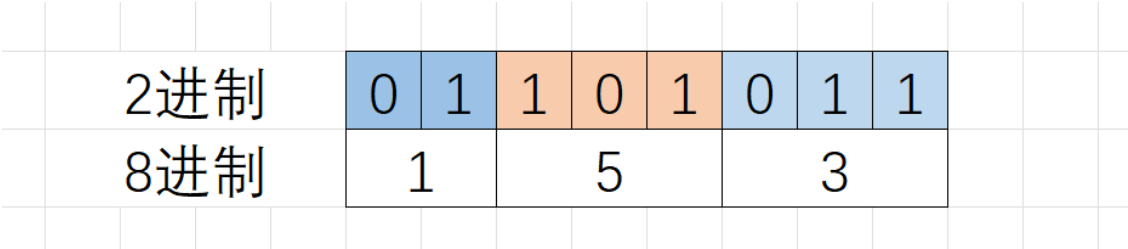
10进制转2进制

2.2 2进制转8进制和16进制

2.2.1 2进制转8进制

8进制的数字每一位是 0~7 的，0~7 的数字，各自写成2进制，最多有3个2进制位就足够了，比如7的二进制是111，所以在2进制转8进制数的时候，从2进制序列中右边低位开始向左每3个2进制位会换算一个8进制位，剩余不够3个2进制位的直接换算。

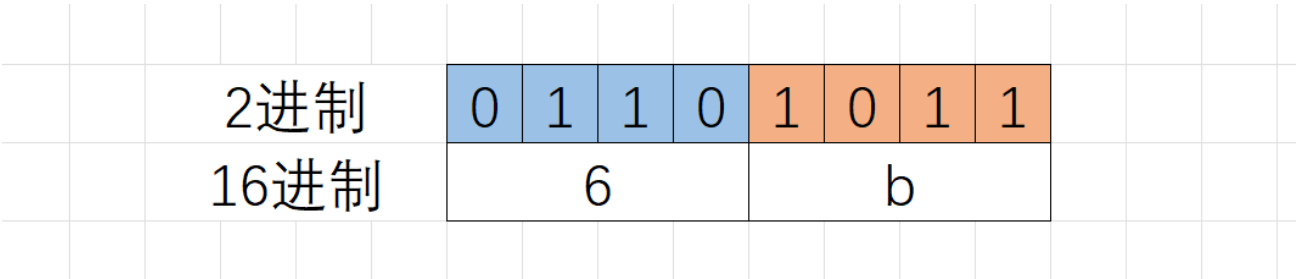
如：2进制的 01101011，换成8进制：0153，0开头的数字，会被当做8进制。



2.2.2 2进制转16进制

16进制的数字每一位是0~9, a~f 的，0~9, a~f的数字，各自写成2进制，最多有4个2进制位就足够了，比如 f 的二进制是1111，所以在2进制转16进制数的时候，从2进制序列中右边低位开始向左每4个2进制位会换算一个16进制位，剩余不够4个二进制位的直接换算。

如：2进制的01101011，换成16进制：0x6b，16进制表示的时候前面加0x

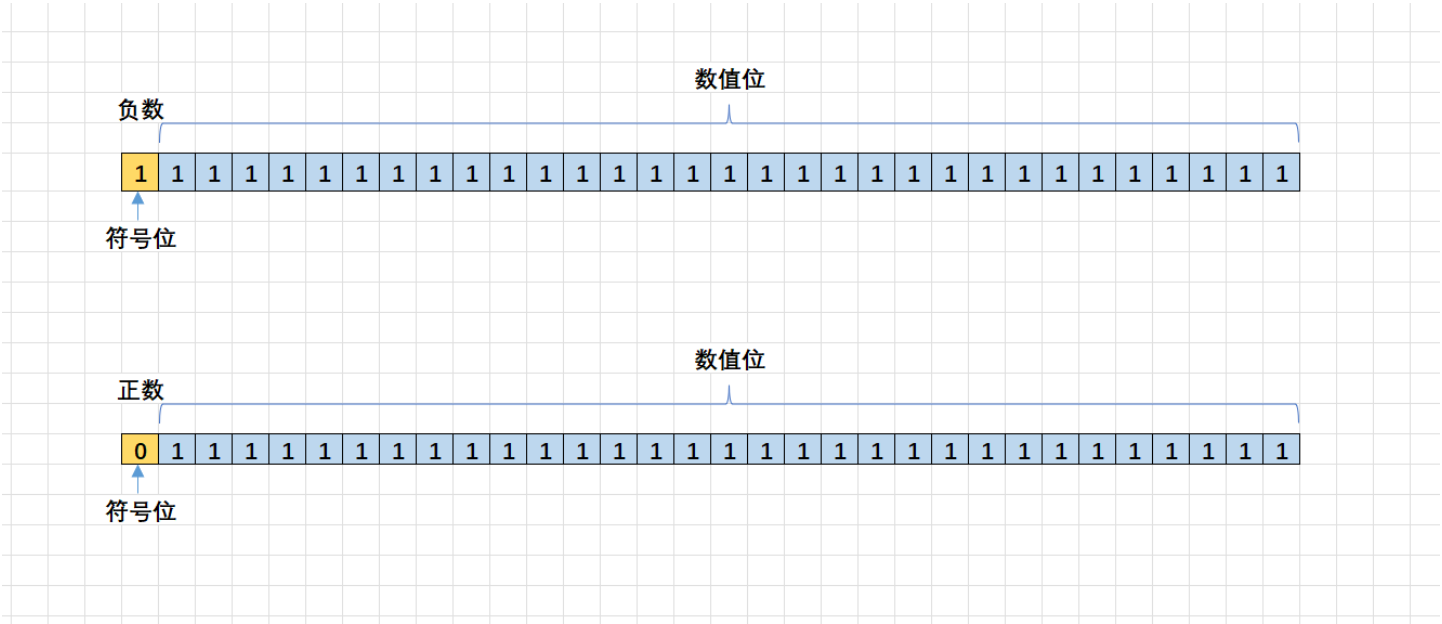


3. 原码、反码、补码

整数的2进制表示方法有三种，即原码、反码和补码

有符号整数的三种表示方法均有**符号位**和**数值位**两部分，2进制序列中，最高位的1位是被当做符号位，剩余的都是数值位。

符号位都是用0表示“正”，用1表示“负”。



正整数的原、反、补码都相同。

负整数的三种表示方法各不相同。

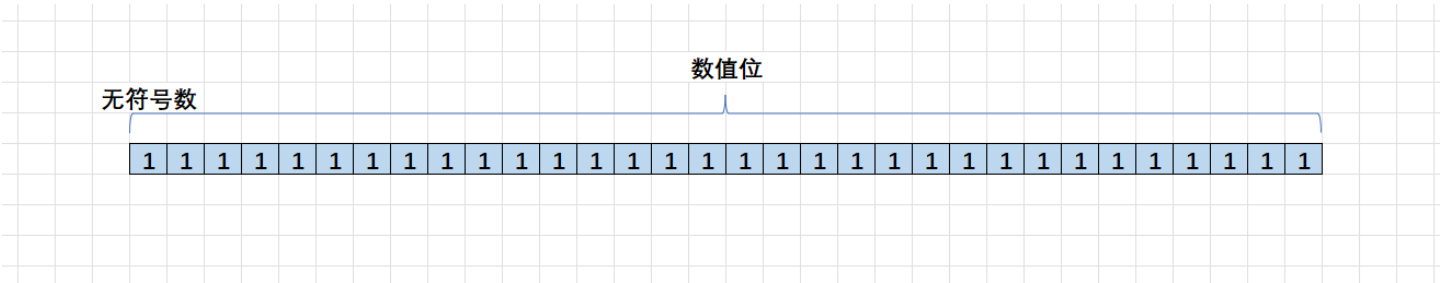
原码：直接将数值按照正负数的形式翻译成二进制得到的就是原码。

反码：将原码的符号位不变，其他位依次按位取反就可以得到反码。

补码：反码+1就得到补码。

补码得到原码也是可以使用：取反，+1的操作。

无符号整数的三种 `2` 进制表示相同，没有符号位，每一位都是数值位。



对于整形来说：数据存放内存中其实存放的是补码。

为什么呢?

> 在计算机系统中，数值一律用补码来表示和存储。原因在于，使用补码，可以将符号位和数值域统一处理；同时，加法和减法也可以统一处理（**CPU只有加法器**）此外，补码与原码相互转换，其运算过程是相同的，不需要额外的硬件电路。

4. 移位操作符

<< 左移操作符

>> 右移操作符

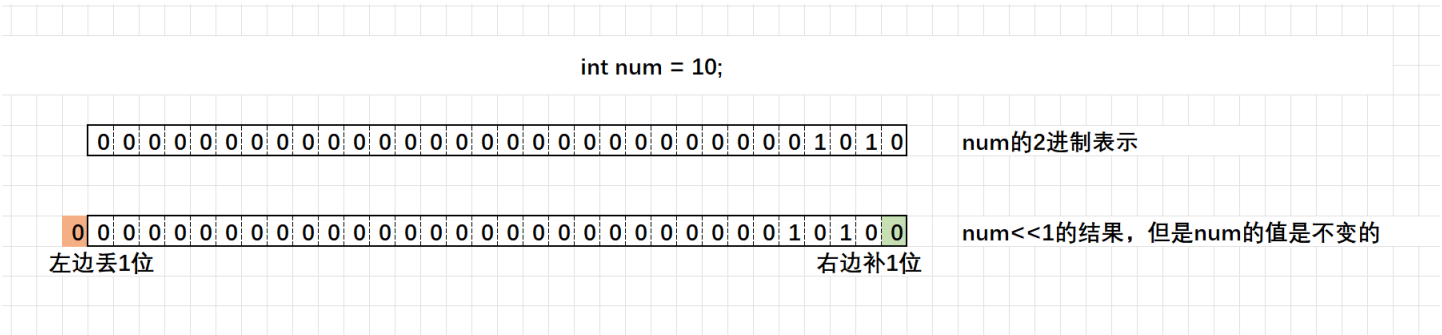
注：移位操作符的操作数只能是整数。

4.1 左移操作符

移位规则：左边抛弃、右边补0

代码块

```c
#include <stdio.h>
int main()
{
    int num = 10;
    int n = num << 1;
    printf("n = %d\n", n);
    printf("num = %d\n", num);
    return 0;
}
```



左移操作符演示

4.2 右移操作符

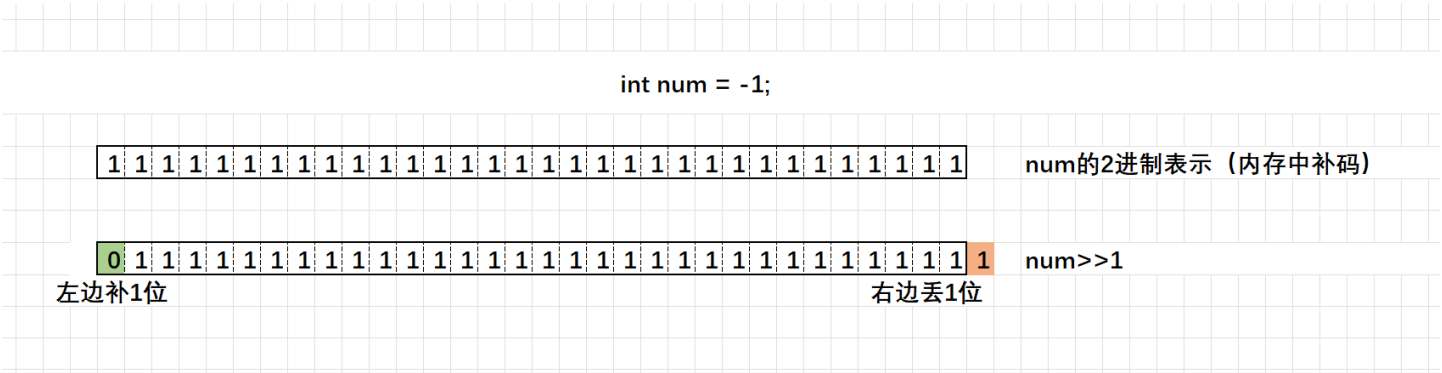
移位规则：首先右移运算分两种：

1. 逻辑右移：左边用0填充，右边丢弃
2. 算术右移：左边用原该值的符号位填充，右边丢弃

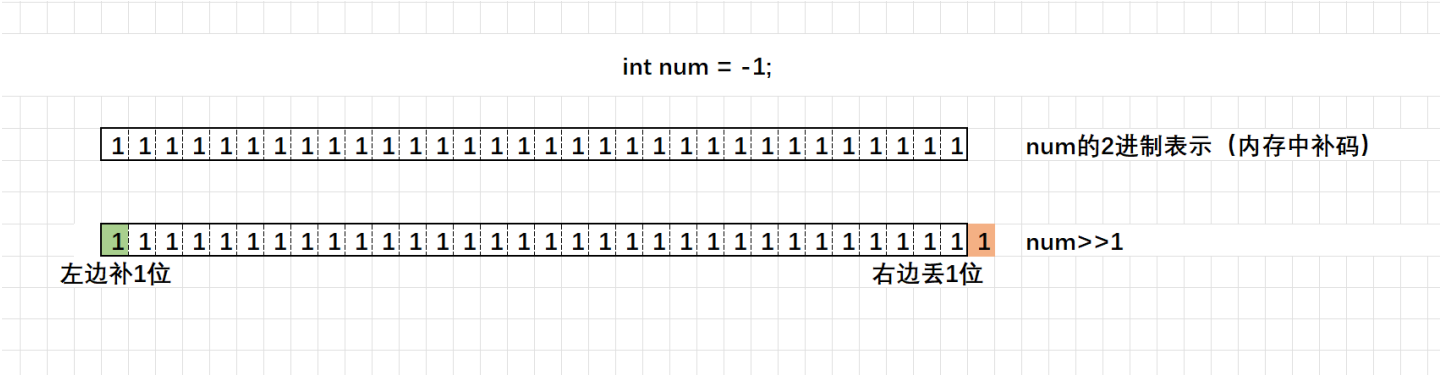
代码块

```c
#include <stdio.h>
int main()
```

```c
{
    int num = 10;
    int n = num >> 1;
    printf("n = %d\n", n);
    printf("num = %d\n", num);
    return 0;
}
```



逻辑右移1位演示



算术右移1位演示

警告⚠：对于移位运算符，不要移动负数位，这个是标准未定义的。

例如：

代码块

```c
int num = 10;
num >> -1;//error
```

5. 位操作符：&、|、^、~

位操作符有：

代码块

```c
&       //按位与
```

```
|         //按位或
^         //按位异或
~         //按位取反
```

注：他们的操作数必须是整数。

直接上代码：

代码块

```c
#include <stdio.h>
int main()
{
    int num1 = -3;
    int num2 = 5;
    printf("%d\n", num1 & num2);
    printf("%d\n", num1 | num2);
    printf("%d\n", num1 ^ num2);
    printf("%d\n", ~0);
    return 0;
}
```

一道变态的面试题：

> 不能创建临时变量（第三个变量），实现两个整数的交换。

代码块

```c
#include <stdio.h>
int main()
{
    int a = 10;
    int b = 20;
    a = a ^ b;
    b = a ^ b;
    a = a ^ b;
    printf("a = %d  b = %d\n", a, b);
    return 0;
}
```

练习1：编写代码实现：求一个整数存储在内存中的二进制中1的个数。

代码块

参考代码：

```c
//方法1
#include <stdio.h>
int main()
{
    int num = 10;
    int count = 0;//计数
    while(num)
    {
        if(num % 2 == 1)
            count++;
        num = num / 2;
    }
    printf("二进制中1的个数 = %d\n", count);
    return 0;
}
//思考这样的实现方式有没有问题?

//方法2:
#include <stdio.h>
int main()
{
    int num = -1;
    int i = 0;
    int count = 0;//计数
    for(i = 0; i < 32; i++)
    {
        if( num & (1 << i) )
            count++;
    }
    printf("二进制中1的个数 = %d\n", count);
    return 0;
}
//思考还能不能更加优化，这里必须循环32次的。

//方法3:
#include <stdio.h>
int main()
{
    int num = -1;
    int i = 0;
    int count = 0;//计数
    while(num)
    {
        count++;
```

```
46            num = num & (num - 1);
47        }
48        printf("二进制中1的个数 = %d\n", count);
49        return 0;
50    }
51    //这种方式是不是很好？达到了优化的效果，但是难以想到。
```

练习2：二进制位置0或者置1

编写代码将13二进制序列的第5位修改为1，然后再改回0

代码块

```
13的2进制序列：  00000000000000000000000000001101
将第5位置为1后：00000000000000000000000000011101
将第5位再置为0：00000000000000000000000000001101
```

参考代码：

代码块

```
#include <stdio.h>
int main()
{
    int a = 13;
    a = a | (1 << 4);
    printf("a = %d\n", a);
    a = a & ~(1 << 4);
    printf("a = %d\n", a);
    return 0;
}
```

6. 单目操作符

单目操作符有这些：

`! 、++、--、&、*、+、-、~ 、sizeof、(类型)`

单目操作符的特点是只有一个操作数，在单目操作符中只有 **&** 和 ***** 没有介绍，这2个操作符，我们放在学习指针的时候学习。

7. 逗号表达式

代码块

```
exp1, exp2, exp3, …expN
```

逗号表达式，就是用逗号隔开的多个表达式。

逗号表达式，从左向右依次执行。整个表达式的结果是最后一个表达式的结果。

代码块

```
//代码1
int a = 1;
int b = 2;
int c = (a > b, a = b + 10, a, b = a + 1);//逗号表达式
c是多少?

//代码2
if (a = b + 1, c = a / 2, d > 0)

//代码3
a = get_val();
count_val(a);
while (a > 0)
{
      //业务处理
      //...
    a = get_val();
    count_val(a);
}

如果使用逗号表达式，改写:
while (a = get_val(), count_val(a), a>0)
{
      //业务处理
}
```

8. 下标访问[]、函数调用()

8.1 [ ] 下标引用操作符

操作数：一个数组名 + 一个索引值(下标)

```
代码块
1   int arr[10];//创建数组
2   arr[9] = 10;//实用下标引用操作符。
3   [ ]的两个操作数是arr和9。
```

8.2 函数调用操作符

接受一个或者多个操作数：第一个操作数是函数名，剩余的操作数就是传递给函数的参数。

```
代码块
1   #include <stdio.h>
2   void test1()
3   {
4       printf("hehe\n");
5   }
6   void test2(const char *str)
7   {
8       printf("%s\n", str);
9   }
10  int main()
11  {
12      test1();            //这里的()就是作为函数调用操作符。
13      test2("hello bit.");//这里的()就是函数调用操作符。
14      return 0;
15  }
```

9. 结构成员访问操作符

9.1 结构体

C语言已经提供了内置类型，如：char、short、int、long、float、double等，但是只有这些内置类型还是不够的，假设我想描述学生，描述一本书，这时单一的内置类型是不行的。

描述一个学生需要名字、年龄、学号、身高、体重等；

描述一本书需要书名、作者、出版社、定价等。C语言为了解决这个问题，增加了结构体这种自定义的数据类型，让程序员可以自己创造适合的类型。

> 📌 结构是一些值的集合，这些值称为成员变量。结构的每个成员可以是不同类型的变量，如：标量、数组、指针，甚至是其他结构体。

9.1.1 结构的声明

```
代码块
```

```
struct tag
{
    member-list;
}variable-list;
```

描述一个学生：

代码块

```
struct Stu
{
    char name[20];//名字
    int age;//年龄
    char sex[5];//性别
    char id[20];//学号
}; //分号不能丢
```

9.1.2 结构体变量的定义和初始化

代码块

```
//代码1：变量的定义
struct Point
{
    int x;
    int y;
}p1;                //声明类型的同时定义变量p1
struct Point p2;    //定义结构体变量p2

//代码2:初始化。
struct Point p3 = {10, 20};

struct Stu          //类型声明
{
    char name[15];//名字
    int age;      //年龄
};

struct Stu s1 = {"zhangsan", 20};//初始化
struct Stu s2 = {.age=20, .name="lisi"};//指定顺序初始化

//代码3
struct Node
{
```

```
    int data;
    struct Point p;
    struct Node* next;
}n1 = {10, {4,5}, NULL};            //结构体嵌套初始化

struct Node n2 = {20, {5, 6}, NULL};//结构体嵌套初始化
```

9.2 结构成员访问操作符

9.2.1 结构体成员的直接访问

结构体成员的直接访问是通过点操作符（.）访问的。点操作符接受两个操作数。如下所示：

代码块

```
#include <stdio.h>
struct Point
{
    int x;
    int y;
}p = {1,2};

int main()
{
    printf("x: %d y: %d\n", p.x, p.y);
    return 0;
}
```

使用方式：结构体变量.成员名

9.2.2 结构体成员的间接访问

有时候我们得到的不是一个结构体变量，而是得到了一个指向结构体的指针。如下所示：

代码块

```
#include <stdio.h>
struct Point
{
    int x;
    int y;
};

```

```c
int main()
{
    struct Point p = {3, 4};
    struct Point *ptr = &p;
    ptr->x = 10;
    ptr->y = 20;
    printf("x = %d y = %d\n", ptr->x, ptr->y);
    return 0;
}
```

使用方式：结构体指针->成员名

综合举例：

代码块

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

struct Stu
{
    char name[15];//名字
    int age;      //年龄
};

void print_stu(struct Stu s)
{
    printf("%s %d\n", s.name, s.age);
}

void set_stu(struct Stu* ps)
{
    strcpy(ps->name, "李四");
    ps->age = 28;
}

int main()
{
    struct Stu s = { "张三", 20 };
    print_stu(s);
    set_stu(&s);
    print_stu(s);
    return 0;
}
```

更多关于结构体的知识，后期在《第20讲：自定义类型：结构体》中讲解。

10. 操作符的属性：优先级、结合性

C语言的操作符有2个重要的属性：优先级、结合性，这两个属性决定了表达式求值的计算顺序。

10.1 优先级

优先级指的是，如果一个表达式包含多个运算符，哪个运算符应该优先执行。各种运算符的优先级是不一样的。

```
代码块
1   3 + 4 * 5;
```

上面示例中，表达式 `3 + 4 * 5` 里面既有加法运算符（`+`），又有乘法运算符（`*`）。由于乘法的优先级高于加法，所以会先计算 `4 * 5`，而不是先计算 `3 + 4`。

10.2 结合性

如果两个运算符优先级相同，优先级没办法确定先计算哪个了，这时候就看结合性了，则根据运算符是左结合，还是右结合，决定执行顺序。大部分运算符是左结合（从左到右执行），少数运算符是右结合（从右到左执行），比如赋值运算符（`=`）。

```
代码块
1   5 * 6 / 2;
```

上面示例中，`*` 和 `/` 的优先级相同，它们都是左结合运算符，所以从左到右执行，先计算 `5 * 6`，再计算 `/ 2`。

运算符的优先级顺序很多，下面是部分运算符的优先级顺序（按照优先级从高到低排列），建议大概记住这些操作符的优先级就行，其他操作符在使用的时候查看下面表格就可以了。

- 圆括号（`()`）
- 自增运算符（`++`），自减运算符（`--`）
- 单目运算符（`+` 和 `-`）
- 乘法（`*`），除法（`/`）
- 加法（`+`），减法（`-`）

- 关系运算符（`<`、`>`等）
- 赋值运算符（`=`）

由于圆括号的优先级最高，可以使用它改变其他运算符的优先级。

优先级	运算符	描述	结合性

1	`++ --`
	`()`
	`[]`
	`.`
	`->`
	`(type){list}`
		后缀自增与自减
		函数调用
		数组下标
		结构体与联合体成员访问
		结构体与联合体成员通过指针访问
		复合字面量(C99)
			从左到右

2	`++ --`
	`+ -`
	`! ~`
	`(type)`
	`*`
	`&`
	`sizeof`
	`_Alignof`
		前缀自增与自减[注 1]
		一元加与减
		逻辑非与逐位非
		转型
		间接（解引用）
		取址
		取大小[注 2]
		对齐要求(C11)
			从右到左

3	`* / %`	乘法、除法及余数	从左到右
4	`+ -`	加法及减法	
5	`<< >>`	逐位左移及右移	
6	`< <=` `> >=`	分别为 < 与 ≤ 的关系运算符 分别为 > 与 ≥ 的关系运算符	
7	`== !=`	分别为 = 与 ≠ 关系	
8	`&`	逐位与	
9	`^`	逐位异或（排除或）	
10	`|`	逐位或（包含或）	
11	`&&`	逻辑与	
12	`||`	逻辑或	
13	`?:`	三元条件[注 3]	从右到左

14[注 4]	`=`
	`+= -=`
	`*= /= %=`
	`<<= >>=`
	`&= ^= |=`
		简单赋值
		以和及差赋值
		以积、商及余数赋值
		以逐位左移及右移赋值
		以逐位与、异或及或赋值	

15	`,`	逗号	从左到右

参考：https://zh.cppreference.com/w/c/language/operator_precedence

11. 表达式求值

11.1 整型提升

C语言中整型算术运算总是至少以缺省（默认）整型类型的精度来进行的。

为了获得这个精度，表达式中的字符和短整型操作数在使用之前被转换为普通整型，这种转换称为**整型提升**。

整型提升的意义：

> 表达式的整型运算要在CPU的相应运算器件内执行，CPU内整型运算器(ALU)的操作数的字节长度一般就是int的字节长度，同时也是CPU的通用寄存器的长度。
>
> 因此，即使两个char类型的相加，在CPU执行时实际上也要先转换为CPU内整型操作数的标准长度。
>
> 通用CPU（general-purpose CPU）是难以直接实现两个8比特字节直接相加运算（虽然机器指令中可能有这种字节相加指令）。所以，表达式中各种长度可能小于int长度的整型值，都必须先转换为int或unsigned int，然后才能送入CPU去执行运算。

代码块

```
//实例1
char a,b,c;
...
a = b + c;
```

b和c的值被提升为普通整型，然后再执行加法运算。

加法运算完成之后，结果将被截断，然后再存储于a中。

如何进行整体提升呢？

> 1. 有符号整数提升是按照变量的数据类型的符号位来提升的，也就是在高位补充符号位
> 2. 无符号整数提升，高位补0

代码块

```
//负数的整形提升
char c1 = -1;
```

```
变量c1的二进制位(补码)中只有8个比特位:
1111111
因为 char 为有符号的 char
所以整形提升的时候，高位补充符号位，即为1
提升之后的结果是:
11111111111111111111111111111111

//正数的整形提升
char c2 = 1;
变量c2的二进制位(补码)中只有8个比特位:
00000001
因为 char 为有符号的 char
所以整形提升的时候，高位补充符号位，即为0
提升之后的结果是:
00000000000000000000000000000001

//无符号整形提升，高位补0
```

11.2 算术转换

如果某个操作符的各个操作数属于不同的类型，那么除非其中一个操作数的转换为另一个操作数的类型，否则操作就无法进行。下面的层次体系称为**寻常算术转换**。

代码块

```
long double
double
float
unsigned long int
long int
unsigned int
int
```

如果某个操作数的类型在上面这个列表中排名靠后，那么首先要转换为另外一个操作数的类型后执行运算。

11.3 问题表达式解析

11.3.1 表达式1

代码块

```
//表达式的求值部分由操作符的优先级决定。
//表达式1
```

```
3   a * b + c * d + e * f
```

表达式1在计算的时候，由于 `*` 比 `+` 的优先级高，只能保证， `*` 的计算是比 `+` 早，但是优先级并不能决定第三个 `*` 比第一个 `+` 早执行。

所以表达式的计算机顺序就可能是：

代码块

```
1   a*b
2   c*d
3   a*b + c*d
4   e*f
5   a*b + c*d + e*f
```

代码块

```
1   a*b
2   c*d
3   e*f
4
5   a*b + c*d
6   a*b + c*d + e*f
```

或者

11.3.2 表达式2

代码块

```
1   //表达式2
2   c + --c;
```

同上，操作符的优先级只能决定自减 `--` 的运算在 `+` 的运算的前面，但是我们并没有办法得知， `+` 操作符的左操作数的获取在右操作数之前还是之后求值，所以结果是不可预测的，是有歧义的。

11.3.3 表达式3

代码块

```
1   //表达式3
2   int main()
3   {
4       int i = 10;
5       i = i-- - --i * ( i = -3 ) * i++ + ++i;
6       printf("i = %d\n", i);
7       return 0;
8   }
```

表达式3在不同编译器中测试结果：非法表达式程序的结果

值	编译器
—128	Tandy 6000 Xenix 3.2
—95	Think C 5.02(Macintosh)
—86	IBM PowerPC AIX 3.2.5
—85	Sun Sparc cc(K&C编译器)
—63	gcc，HP_UX 9.0，Power C 2.0.0
4	Sun Sparc acc(K&C编译器)
21	Turbo C/C++ 4.5
22	FreeBSD 2.1 R
30	Dec Alpha OSF1 2.0
36	Dec VAX/VMS
42	Microsoft C 5.1

11.3.4 表达式4

代码块

```
#include <stdio.h>

int fun()
{
      static int count = 1;
      return ++count;
}
int main()
{
      int answer;
      answer = fun() - fun() * fun();
      printf( "%d\n", answer);//输出多少?
      return 0;
}
```

这个代码有没有实际的问题？**有问题！**

虽然在大多数的编译器上求得结果都是相同的。

但是上述代码 `answer = fun() - fun() * fun();` 中我们只能通过操作符的优先级得知：先算乘法，再算减法。

函数的调用先后顺序无法通过操作符的优先级确定。

11.3.5 表达式5

代码块

```
//表达式5
#include <stdio.h>
int main()
{
    int i = 1;
    int ret = (++i) + (++i) + (++i);
    printf("%d\n", ret);
    printf("%d\n", i);
    return 0;
}
//尝试在linux 环境gcc编译器，VS2013环境下都执行，看结果。
```

gcc编译器执行结果：

```
[root@centos7net test]# ./a.out
10
4
```

VS2022运行结果：

看看同样的代码产生了不同的结果，这是为什么？

简单看一下汇编代码，就可以分析清楚.

这段代码中的第一个 `+` 在执行的时候，第三个++是否执行，这个是不确定的，因为依靠操作符的优先级和结合性是无法决定第一个 `+` 和第三个前置 `++` 的先后顺序。

11.4 总结

即使有了操作符的优先级和结合性，我们写出的表达式依然有可能不能通过操作符的属性确定唯一的计算路径，那这个表达式就是存在潜在风险的，建议不要写出特别复杂的表达式。

完