# 4.1概念

**表达式**:多个运算对象通过运算符获得一个结果

- 运算对象
- 运算符类型(按对象数量分为一元,二元,三元)
- 运算规律(优先级,结合律,求值顺序),用于求取复合表达式(多个运算符),其中**优先级**针对不同运算符,**结合律**规定同种运算符的结合方向,**括号**无视优先级和结合律

**求值顺序**

```
int i=f1()*f2()
```

只知道乘法执行前f1(),f2()两个函数一定调用,但不直到调用顺序,如果两个函数修改同一个对象,会产生未定义的结果

```
cout<<i<<" "<<++i<<endl
```

同理,有可能先求i,也有可能先求i++,行为不可预知

**四种规定求值顺序的运算符**:逻辑与&& 逻辑或|| 条件运算符? : 逗号运算符,

**书写正确的表达式**:没有把握就使用括号,表达式某处改变了一个对象,其他地方就不要使用该对象

**左值和右值**(H)

右值使用的是其内容,用于左值使用的是身份(内存位置).

特殊的,左值当作右值使用的时候,实际使用的也是其内容.

- 赋值运算符要一个(非常量)左值作为左侧对象,得到结果仍是左值
- 取地址符作用于一个左值对象,返回一个指向该对象的指针,这个指针是右值
- 内置解引用,下标运算符(如string和vector),迭代器解引用的求值结果都是左值
- 内置类型和迭代器的**递增递减**运算符作用于左值,获得左值

# 4.2算数运算符

| 运算符(左结合律) | 功能 |
| --- | --- |
| -,+ | 一元负号,一元正号 |
| *,/ | 乘除 |
| % | 取余 |
| +,- | 加法减法 |

作用于任意算术类型以及可转化为算术类型的对象

运算对象和求值结果都是右值

**取余(取模)运算符%**

1. 运算对象必须是整数
2. 除取负导致溢出的情况,负数取余和负数除法类似,都把负号提到外边,不过提之前取余只需要看左边的负号

   -21%-8 // 等价于-(21%8) 结果-5
   21%-5 //等价于(21%5) 结果1

   -21/-8 //等价于(21/8) 结果2
   21/-5 //等价于-(21/5) 结果-4

# 4.3逻辑和关系运算符

| 结合律 | 运算符 | 功能 |
| --- | --- | --- |
| 右 | ! | 逻辑非 |
| 左 | <, <= | 小于, 小于等于 |
| 左 | >, >= | 大于, 大于等于 |
| 左 | ==, != | 相等,不相等 |
| 左 | &&, \|\| | 逻辑与,逻辑或 |

关系运算符作用于算数类型和指针类型
逻辑运算符作用于任意能转换成布尔值的类型
两者返回值都是布尔类型,运算对象和求值结果都是右值.

- **短路运算**:逻辑与(逻辑或)当左边为真(假)才对右侧求值
- i<j<k是错误表示,此处i<j的结果再与k运算,应该用i<j&&j<k
- 测试算数对象或指针对象真值:

```
if(p)
if(!p)
if(p==true) //错误,true提升为1
```

# 4.4赋值运算符

- 左侧对象必须是一个可修改的左值(不能是常量),左值与右值类型不同时右值转化为左值类型
- 赋值运算满足右结合律,多重赋值要满足左对象和右类型相同或能从右转化为左类型:

```
int ival,*pval;
ival=pval=0; 错误,不能把int指针赋给int
```

- 列表初始化:内置类型列表只能包含一个值,而且转化也不应该大于目标类型的空间

```
int k=0;//int型,值0
k=3.14;//int型,值3
k={3.14}//错误,窄化转换
```

- 优先级较低,有时候需要添加括号
- 复合赋值运算符:
  算数运算符:+= -= *= /=
  位运算符:<<= >>= &= ^= |=
  任意复合运算符完全等价于a=a op b,除了不会求值两次(右边表达式一次和左边赋值一次)

# 4.5递增与递减运算符

- 分为前置版本和后置版本如++a, a++,尽量使用前置版本,因为后置版本需要存储原始值增加额外工作量
- 解引用和递增运算符混用:

```
*ptr;
++ptr
```

简化为

```
*ptr++
```

因为后置运算符优先级高于解引用,所以等价为*(ptr++)
ptr++返回初始值的副本,该副本被解引用

- 赋值语句对象可以按任意顺序求值

```
*beg=tou(*beg++);//错误,顺序未知
```

两端都用到了beg,而且改变了beg值

# 4.6成员访问运算符

点运算符和箭头运算符可用于访问成员
点运算符获取类对象的一个成员,箭头运算符ptr->mem等价于(*ptr).mem ,先解引用再访问成员

```
string s1="hello",*p=&s1;
auto n=s1.size()  运行string对象s1的size成员

n=(*p).size()  运行p所指对象的size成员
n=p->size  等价于n=(*p).size()
```

## 4.7条件运算符

1. cond ? exp1 : exp2,首先求cond值若为真对exp1求值并返回该值,否则求exp2返回该值
2. 条件运算符嵌套不要超过两到三层,不然可读性急剧降低
3. 条件运算符优先级非常低,需要加上括号

# 4.8位运算符

作用于整数类型的运算对象,把对象看成二进制位的集合

求反(~),左移(<<),右移(>>),位与(&),位或(|),位异或(^)

1. 建议用位运算符处理无符号类型,当带有负号时,如何处理符号结果依赖于机器
2. 移位运算符(IO运算符)满足**左结合律**,优先级低于算数运算符,高于关系运算符,赋值运算符和条件运算符.重载运算符与之类似.

## 4.9sizeof运算符

返回一个表达式或一个类型所占的字节数,形式如下:

- sizeof (type)
- sizeof expr **返回表达式结果类型的大小**

sizeof第二种情况下并不计算expr,优先级与*相同,满足右结合律.

sizeof不一定后面要括号,所以sizeof不是函数,sizeof本身在编译后消失,只有表达式生成的数值,sizeof类似于编译预处理

sizeof运算符的结果部分依赖于其作用的类型:

- 对char或者char类型的表达式使用,结果1
- 对引用类型使用得到引用对象类型的大小
- 对指针使用得到**指针本身**所占空间大小
- 对解引用指针使用得到指针所指对象空间大小,指针不需有效(sizeof不计算,所以无需指针初始化,因为不会使用所以安全)
- 数组进行使用得到**整个数组**大小
- **对string和vector对象只返回该类型固定部分的大小,不会计算对象的元素占用了多少空间**(存储在堆上,动态分配)

## 4.10逗号运算符

含有两个运算对象,从左到右依次求值,左侧舍弃,**实际结果为右侧**.常用于for语句的表达式中

# 4.11类型转换

隐式转换(implicit convertion)指类型自动完成,不需要程序员介入
下列情况编译器自动转换运算对象类型:

1. 大多数表达式中,比int小的整型首先提升为较大整型
2. 条件中,非布尔值转化为布尔值
3. 初始化,初始值转化成变量的类型;赋值,右侧转化为左侧
4. 算数运算或关系运算的对象有多种类型,需要转化成同一类型
5. 函数调用有时候也会发生类型转化

### 4.11.1算术转换

一种算数类型转化成另一种算数类型,规则:

1. 运算对象转化成最宽
2. 有符号转化为无符号
3. 整型和浮点型都有就转化成浮点型

### 4.11.2其他隐式转换

数组转化为指针:

```
int ia[10];
int *ip=ia;  ia转化为指向首元素的指针
```

数组被用作decltype关键字的参数,或者作为取地址符,sizeof的运算对象时,上述转换不会发生,同样,用引用来初始化一维数组,转换也不会发生

指针的转换: 常量0和nullptr能转化成任意指针类型;指向任意非常量的指针能转化成*void;指向任意对象的指针能转化成const *void.

转化成布尔类型:指针或算数值为0转化为false,否则转化为true

转化成常量:允许将指向非常量类型的指针转化成常量类型的指针,对于引用也是这样,反之则不行,因为试图删除底层const:

```
int i;
const int &j=i;
const int *p=&i;
int &r=j,*q=p; 错误,不允许const转化成非常量
```

类类型的转换:比如string s="value",字符串字面值转化成string类型,但是不能一次转化多个类型

### 4.11.3显式转换

显式的将对象转化成另一种类型,称为强制类型转换(cast)

cast-name<type>(expr)

- type:转换类型
- expr:转换的值
- cast-name是static_cast,dynamic_cast,const_cast,reinterment_cast中的一个,dynamic_cast支持运行时识别,cast_name规定了执行哪种转换

**static_cast**

不包含底层const,就可以使用static_cast

当要把一个较大算术类型赋值给较小的类型时,可以忽略掉精度损失防止编译器警告

也可以用于编译器无法自动执行的类型转换,比如*void转化为*double

**const_cast**

改变运算对象的底层const,将常量类型改为其他类型称为去掉const性质(cast away the const),只有const_cast能改变表达式常量类型,其他方式都会出错:

```
const char *pc;
char *p=const_cast<char*>(pc)
```

**reinterment_cast**(风险大,少用)

为运算对象的位模式提供重新解释,使reinterment_cast很危险,需要了解类型和编译器

# 拓展forward declaration and ODR

```
int main(){
    add();
    return 0;
}
void add(){
}
```

错误原因:编译器顺序按顺序编译代码文件,add()定义在main()之后
解决方法:

1. 重新排序:把add()代码放到main()前
2. 前向声明:forward declaration

前向声明: 包括返回类型,名字,参数类型.可选的参数名,**但不含函数体** .同一个函数声明,定义,调用的参数类型和数量要一致

定义之前告知编译器标识符的存在,即使不知道在哪也不会报错

如果前向声明一个函数但是未定义会发生什么,两种情况:

1. 未调用该函数,正常编译和运行
2. 调用该函数,正常编译,但链接失败

## ODR(one definition rule)

1. 在给定的**文件**中，函数、变量、类型或模板只能有一个定义。(声明可以多个)
2. 在给定的**程序**中，一个**变量**或**普通函数**只能有一个定义。之所以有这种区别，是因为程序可以有多个文件
3. 类型、模板、**内联**函数和**内联**变量可以在不同的文件中具有相同的定义。

- 违反ODR的第1部分将导致编译器发出重定义错误。违反ODR第2部分可能会导致链接器发出重定义错误。违反ODR第3部分将导致未定义的行为。