## 基础：

函数调用也是一种特殊的运算符。

对于左值和右值，当一个对象被用作右值的时候，用的是对象的值（内容）；当对象被用作左值的时候，用的是对象的身份（在内存中的位置）。在需要右值的地方可以用左值来代替，但是不能把右值当成左值使用。

1. 赋值运算符需要一个左值作为左侧运算对象，得到的结果也仍然是一个左值，
2. 取地址符作用于一个左值运算对象，返回一个指向该运算对象的指针，这个指针是右值，
3. 内置解引用运算符、下标运算符、迭代器解引用运算符、STL容器的下标运算符的求值结果是左值，
4. 内置类型和迭代器的递增递减运算符作用于左值运算对象，其前置版本所得的结果也是左值。

使用关键字decltype的时候，左值和右值也有所不同，如果表达式是左值，decltype得到的是一个引用类型。

### 1.1优先级与结合律：

### 1.2求值的顺序：

对于没有指定执行顺序的运算符来说，如果表达式指向并修改了同一对象，将会引发错误并产生未定义的行为。

int i = 0;

cout<<i<<" "<<++i<<endl;//未定义的

有4中运算符明确规定了运算对象的求值顺序：

1. &&运算符：先求左侧对象的值，当左侧为真时再求右侧对象的值（短路求值）。
2. ||运算符: 先求左侧对象的值，当左侧为假时再求右侧对象的值（短路求值）。
3. ?: 运算符：
4. , 运算符：

## 算术运算符：

bool b = true;

bool b2 = -b;//b2是true！

小心算术运算异常（比如说除0）和溢出（超过范围）的情况。

参与取余（%）运算的运算对象必须是整数类型。C++11规定商一律向0取整。m%(-n)等价于m%n，(-m)%n等价于-(m%n)。

## 3.逻辑和关系运算符：

关系运算符连写在一起会产生意想不到的结果。

If(I < K < J)//拿i<k的布尔值和j比较

If(!val)//编译器会把val转换为bool值

## 4.赋值运算符：

赋值运算符的结果是它的左侧运算对象，并且是一个左值。相应的，结果的类型就是左侧运算对象的类型。

如果左侧对象是内置类型，那么初始值列表最多只能包含一个值，而且该值即使转换的话其所占空间也不应该大于目标类型的空间。

int k = {3.14};//错误，窄化转换

int ival = jval = 0;

因为赋值运算符满足右结合律，所以靠右的赋值运算作为靠左的赋值运算符的右侧对象/又因为赋值运算返回的是其左侧运算对象，所以靠右的赋值运算结果被赋给了ival。

## 5.递增和递减运算符：

都必须作用于左值运算对象，前置版本将对象本身作为左值返回，后置版本则将对象原始值的副本作为右值返回。除非必须，否则不用递增递减运算符的后置版本，如果我们不需要修改前的值，那么后置版本就是浪费。

auto pbeg = v.begin();

while(pbeg != v.end() && \*beg >= 0)

cout<<\*pbeg++<<endl;//pbeg++把pbeg的值增加1，然后返回pbeg的初始值的副本作为其求值结果

## 6.成员访问运算符：

表达式ptr->mem等价于(\*ptr).mem。箭头运算符作用于一个指针类型的运算对象，结果是一个左值。点运算符分成两种情况：如果成员所属的对象是左值，那么结果是左值；反之，如果成员所属的对象是右值，那么结果是右值。

## 7.条件运算符：

当运算符的两个表达式都是左值或者都能转换成同一种左值类型时，运算的结果是左值，否则运算的结果是右值。允许在条件运算符的内部嵌套另外一个条件运算符。

string finalgrade = (grade < 60)?"fail":"pass";

## 8.位运算符：

|  |  |  |
| --- | --- | --- |
| 表4.3：位运算符（左结合律） | | |
| 运算符 | 功能 | 用法 |
| ~ | 位求反 | expr |
| < | 左移 | expr1 << expr2 |
| > | 右移 | expr1 >> expr2 |
| & | 位与 | expr & expr |
| ^ | 位异或 | expr ^ expr |
| | | 位或 | expr | expr |

如果运算对象是小整型，则它的值会被自动提升成较大的整数类型，如果运算对象是带符号的且它的值为负，那么位运算符如何处理运算对象的“符号位”依赖于机器。而且此时的左移操作可能会改变符号位的值，因此是一种未定义的行为。

### 8.1移位运算符：

右侧的运算对象一定不能为负，而且值必须严格小于结果的位数，否则就会产生未定义的行为。移出边界之外的就舍弃掉了。

左移运算符在右侧插入值为0的二进制位；右移运算符则依赖于其左侧运算对象的类型：如果该运算对象是无符号类型，在左侧插入值为0的二进制位，如果是有符号，在左侧插入符号位的副本或值为0的二进制位。

### 8.2位求反运算符：

位求反之前先做提升运算，提升时对象原来的位保持不变，往高位添加0即可。求反别忘了把提升运算后产生的0取反！

### 8.3位与运算符：同1则1

### 8.4位或运算符：有1则1

### 8.5位异或运算符：只有一个为1则1

## 9.sizeof运算符：

Sizeof运算符满足右结合律，其所得的值是一个size\_t类型的常量表达式。运算符的运算对象有两种形式：

Sizeof(type);

Sizeof expr;//返回表达式结果类型的大小，sizeof并不计算其实际对象的值

Int \*p;

Sizeof p;//指针所占空间的大小

Sizeof \*p;//p所指类型的空间的大小

对解引用指针执行sizeof运算得到指针指向的对象所占空间的大小，指针不需要有效。

对string对象或vector对象执行sizeof运算只返回该类型固定部分的大小，不会计算对象中得到数组中元素的个数。

Int ia[10];

Sizeof(ia)/sizeof(\*ia);//返回ia的元素数量

注意：

32位编译器（加不加unsigned并没有卵用）：

bool:1个字节

unsigned char ：1个字节

char ：1个字节

wchar\_t:2个字节

char16\_t:2个字节

char32\_t:4个字节

unsigned short:2个字节

short: 2个字节

unsigned int : 4个字节

int： 4个字节

unsigned long: 4个字节

long: 4个字节

long long: 8个字节

float: 4个字节

double: 8个字节

long double:12个字节

char\*: 4个字节

64位编译器：

char\*: 8个字节

long: 8个字节

unsigned long: 8个字节

## 10.逗号运算符：

含有两个运算对象，首先对左侧的表达式求值，然后将左侧结果丢弃，真正的结果是右侧表达式的值，如果右侧运算对象是左值，那么最终的求值结果也是左值。

## 11.类型转换：

如果两种类型有关联，就可以互相类型转换。

Int val = 3.541 + 3;//先根据类型转换规则设法将运算对象的类型统一后再求

### 11.1隐式转换：

整型提升：只要所有的值能存在int里，就会提升成int，否则，提升成unsigned int。转换后的类型要能容纳原类型所有可能的值，并且尽量选择较小的类型。对于小数，一概切除。

首先执行整体提升。

1. 如果类型匹配，无需进行进一步的转换。
2. 如果两个运算对象的类型要么都是带符号的、要么都是无符号的，则小类型的运算对象转换成较大类型。
3. 如果两个运算对象的类型一个是带符号、一个是无符号，但是其中的无符号类型大小不小于带符号类型，那么带符号的运算对象转换成无符号的。（如unsigned int a = 1;int b = -1; 会取余）。
4. 如果两个运算对象的类型一个是带符号、一个是无符号，但是带符号类型大于无符号类型，此时转换的结果依赖于机器，如果无符号类型的所有值都能存在该带符号类型中，则无符号类型的运算对象转换成带符号类型，如果不能，那么带符号类型的运算对象转换成无符号类型。

### 11.2其他隐式转换：

①数组转为指针：

②指针的转换：常量整数值0或者字面值nullptr能转换成任意指针类型；指向任意非常量的指针能转为void\*；指向任意对象的指针能转成const \*void

③算数类型或指针转换成布尔型：

④指向非常量类型的指针/引用转换成指向相应的常量指针/引用；反之不行，因为它试图删除底层const

⑤类类型定义的转换：在需要标准库string类型的地方使用C风格字符串；在循环条件部分读入istream。

### 11.3显式转换：

格式：cast-name<type>(expression); type是转换的目标类型，expression是要转换的值。如果type是引用，则结果是左值。cast-name是static\_cast、dynamic\_cast、const\_cast和reinterpret\_cast中的一种。

①dynamic\_cast：支持运行时类型识别。

②static\_cast:任何不包含底层const的类型转换都可以使用.

void\* p = &d;//正确，任何非常量对象的地址都能存入void\*

double \*dp = static\_cast<double\*>(p);//正确，将void转换回初始的指针类型，找回存在于void\*指针中的值，但要注意我们必须确保转换后的类型就是指针所指的类型，类型一旦不符，将产生未定义的后果。

③const\_cast：只能改变对象的底层const。

const char \*pc;

char \*p = const\_cast<char\*>(pc);//正确，但是通过p写值是未定义的行为。如果对象本事不是一个常量，使用强制类型转换获得写权限是合法的行为，然而如果对象是一个常量，再使用const\_cast执行写操作就会产生未定义的后果。

const char \*cp;

char \*q = static\_cast<char\*>(cp);//static\_cast不能换掉非const

static\_cast<string>(cp);//正确

const\_cast<string>(cp);//错误，const\_cast只改变常量属性

④reinterpret\_cast:为运算对象的位模式提供较低层次上的重新解释。使用是十分危险的，类型改变了，但是编译器没有给出任何警告或者错误的提示。

int \*ip;

char \*pc = reinterpret\_cast<char\*>(ip);//pc是一个int而不是char

### 11.4旧式的强制类型转换：

有两种形式：

①type(expr);//函数形式的强制类型转换

②(type) expr;//C语言风格的强制类型转换

## 12.运算符优先级表：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 结合律和运算符 | | 功能 | 用法 | 参考页码 |
| 左 | :: | 全局作用域 | ::name | 286 |
| 左 | :: | 类作用域 | class::name | 88 |
| 左 | :: | 命名空间作用域 | namespace::name | 82 |
| 左 | . | 成员选择 | object.member | 23 |
| 左 | -> | 成员选择 | pointer->member | 110 |
| 左 | [] | 下标 | expr[expr] | 116 |
| 左 | () | 函数调用 | name(expr\_list) | 23 |
| 左 | () | 类型构造 | type(expr\_list) | 164 |
| 右 | ++ | 后置递增运算 | lvalue++ | 147 |
| 右 | -- | 后置递减运算 | lvalue-- | 147 |
| 右 | typeid | 类型ID | typeid(type) | 826 |
| 右 | typeid | 运行时类型ID | typeid(expr) | 826 |
| 右 | explicit cast | 类型转换 | *cast\_name*<type>(expr) | 162 |
| 右 | ++ | 前置递增运算 | ++lvalue | 147 |
| 右 | -- | 前置递减运算 | --lvalue | 147 |
| 右 | ~ | 位求反 | ~expr | 152 |
| 右 | ! | 逻辑非 | !expr | 141 |
| 右 | - | 一元负号 | -expr | 140 |
| 右 | + | 一元正号 | +expr | 140 |
| 右 | \* | 解引用 | \*expr | 53 |
| 右 | & | 取地址 | &lvalue | 52 |
| 右 | () | 类型转换 | (type) expr | 164 |
| 右 | sizeof | 对象的大小 | sizeof expr | 156 |
| 右 | sizeof | 类型的大小 | sizeof(type ) | 156 |
| 右 | Sizeof… | 参数包的大小 | sizeof...( name ) | 700 |
| 右 | new | 创建对象 | new type | 458 |
| 右 | new[] | 创建数组 | new type[size] | 458 |
| 右 | delete | 释放对象 | delete expr | 460 |
| 右 | delete[] | 释放数组 | delete[] expr | 460 |
| 右 | noexcept | 能否抛出异常 | noexcept ( expr ) | 780 |
| 左 | ->\* | 指向成员选择的指针 | ptr->\*ptr\_to\_member | 837 |
| 左 | .\* | 指向成员选择的指针 | obj.\*ptr\_to\_member | 837 |
| 左 | \* | 乘法 | expr \* expr | 139 |
| 左 | / | 除法 | expr / expr | 139 |
| 左 | % | 取模（取余） | expr % expr | 139 |
| 左 | + | 加法 | expr + expr | 139 |
| 左 | - | 减法 | expr - expr | 139 |
| 左 | << | 向左移位 | expr << expr | 152 |
| 左 | >> | 向右移位 | expr >> expr | 152 |
| 左 | < | 小于 | expr < expr | 141 |
| 左 | <= | 小于等于 | expr <= expr | 141 |
| 左 | > | 大于 | expr > expr | 141 |
| 左 | >= | 大于等于 | expr >= expr | 141 |
| 左 | == | 相等 | expr == expr | 141 |
| 左 | != | 不相等 | expr != expr | 141 |
| 左 | & | 位与 | expr & expr | 152 |
| 左 | ^ | 位异或 | expr ^ expr | 152 |
| 左 | | | 位或 | expr | expr | 152 |
| 左 | && | 逻辑与 | expr && expr | 141 |
| 左 | || | 逻辑或 | expr || expr | 141 |
| 右 | ? : | 条件 | expr ? expr : expr | 151 |
| 右 | = | 赋值 | lvalue = expr | 144 |
| 右 | \*=, /=, %= | 复合赋值 | lvalue += expr等 | 144 |
| 右 | +=, -= | 144 |
| 右 | <<=, >>= | 144 |
| 右 | &=, |=, ^= | 144 |
| 右 | throw | 抛出异常 | throw expr | 193 |
| 左 | , | 逗号 | expr, expr | 157 |