chapter 4 表达式

4.1 基础

1. 作用于一个运算对象的运算符是一元运算符，如取地址符（&）和解引用符（*），作用于两个运算对象的运算符是二元运算符，如相等运算符（==）和乘法运算符（*）,还有作用于三个运算对象的三元运算符。函数调用也是一种特殊的运算符，它对运算对象的数量没有限制。

2. 一些符号既能作为一元运算符也能作为二元运算符，比如*号既可以是解引用，也可以是乘号。

3. 即使运算对象的类型不同，只要他们能被转换，也能使用二元运算符进行计算。例如整数变成浮点数，浮点数转换成整数。**小整数类型（如bool、char、short等）通常会被提升成较大的整数类型，主要是int。**

4. C++表达式要不然是右值（rvalue）要不然就是左值（lvalue），左值可以位于赋值语句的左边，而右值不能。在C++中，一个左值表达式的求值结果是一个对象或者一个函数，而以常量对象为代表的某些左值并不能作为赋值语句的左侧运算对象。**归纳：当一个对象被用作右值时，用的是对象的值（内容），而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）。**左值右值的详解看这篇博客

5. **一个重要的原则：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（也就是位置）来使用。当一个左值被当成右值使用时，实际使用是他的内容（值）。**

6. 以下是一些我们常用的需要用到左值的运算符：

   - 赋值运算符需要有个（非常量）左值作为其左侧运算对象，得到的结果也仍然是一个左值。
   - 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，这个指针是一个右值。
   - 内置引用运算符、下标运算符、迭代器解引用运算符、string和vector的下标运算符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运算对象，其前置版本所得的结果也是左值

7. 使用关键字decltype的时候，如果表达式的求值结果是左值，则得到一个引用类型（假设p是int*类型，decltype(*p)返回的是一个引用 int&），如果取地址运算符生成右值，例如decltype(&p)，那么会返回一个int**，也就是一个指向整形指针的指针。

8. 表达式中的括号无视运算符的优先级。程序员可以使用括号将某个表达式的某个局部括起来使其得到优先运算。

9. 算术运算符满足结合律，也就是当运算符的优先级相同，将按照从左向右的顺序组合运算对象

10. 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个对象**，将会引发错误并产生未定义的行为。例如：

```
int i = 0;
cout << i << " " << i++ << endl; // 未定义的
```

```
// 因为程序可能先执行i++再执行i，也可能先执行i再求i++，因此结果产生了不可预知性
```

11. 有4种运算符明确规定了运算对象的求值顺序。分别是 '&&' ， '||' ， '？：' ， ','这4种

12. 对于表达式的结合律例如f() + g() * h() + j()的表达式中，按照优先级，先求g*h，然后结合律规定求f加g和h的乘积，再加j。如果f\g\h\j是无关函数，他们既不会改变同一对象的状态也不执行IO任务，那么函数的调用顺序不受限制，反之，如果其中某几个函数影响同一对象，则他是一条错误的表达式

4.2 算术运算符

1. 算术运算符的优先级（由上至下优先级降低）：

运算符	功能	用法
+	一元正号	+ expr
-	一元负号	- expr
*	乘法	expr * expr
/	除法	expr / expr
%	求余	expr % expr
+	加法	expr + expr
-	减法	expr - expr

2. 算术运算符能作用于任意算术类型以及任意能转换成为算术类型的类型，**算术运算符的运算对象和求值结果都是右值**。在表达式求值之前，小整数类型的运算对象被提升为较大整数类型，**所有运算对象最终都会转换成同一类型**。

3. 一元正号运算符、加法运算符和减法运算符都能作用于指针。当**一元正号**运算符作用于一个指针或算术值时，返回运算对象值的一个（提升后的）副本。一元负号运算符对运算对象值取负后，返回其（提升后的）副本：

```
int i = 1024;
int k = -i; // k是-1024
bool b = true;
bool b2 = -b; // b2是true！！
```

4. **上述布尔值b2先被转换成int型1，然后加上'-'号后变为-1，然后再转换成布尔值则为1（布尔值只有在0的时候为false），所以尽量避免在运算中使用布尔值。**

5. 算术表达式有可能产生未定义的结果，一部分原因是数学性质本身，例如除以0，另一部分源于计算机的特性，例如溢出，当计算的结果超出该类型所能表示的范围时就会溢出。

6. 整数相除结果还是整数，小数部分将被丢弃。例如21/6是3，21/7也是3。

7. 运算符%是取余或取模运算符。**参与取余（%）运算的运算符必须是整数类型**。

```
int ival = 42;
double dval = 3.14;
ival % 12;   // 正确，结果是6。
ival % dval; // 错误， dval是浮点型
```

8. **c++11新标准规定，商的值一律向0取整。**

9. 对于取模运算符（%）来说，如果 m % n 的运算结果不等于0，则它的符号与m相同。**而 (-m) / n 和 m / (-n) 都等于 -(m / n)，m % (-n) 等于 m % n，(-m) % n 等于 -(m % n)**。

运算	结果
21 % 6	3
21 % 7	0
-21 % -8	-5
21 % -5	1

4.3 逻辑和关系运算符

1. 逻辑运算符包含逻辑非（！）逻辑与（&&）逻辑或（||），关系运算符则包含大于小于等于不等于运算符（>, >=, <, <=, ==, !=），两者的返回类型都是布尔值，值为0的运算对象则为假，否则为真。对于这两类运算符来说，**运算对象和求值结果都是右值**。

2. 逻辑与（&&）和逻辑或（||）当且仅当左侧运算对象无法确定表达式的结果时才会计算右侧运算对象的值，这种策略称为短路求值（short-circuit evaluation）：

    - 逻辑与：当且仅当左侧运算对象为真时才对右侧运算对象求值
    - 逻辑或：当且仅当左侧运算对象为假时才对右侧运算对象求值

3. 因为关系运算符的求值结果是布尔值，所以将几个关系运算符连写在一起会产生意想不到的结果：

```
if( i < j < k){
    // 此表达式会先返回i<j的结果，再用该结果和k作比较
    // 例如i<j为真返回1，则再用1<k来返回下一个结果
}

// 实际上我们想表达的是下式
if( i < j && j < k){

}
```

4.4 赋值运算符

1. 赋值运算符的左侧运算对象必须是一个可修改的左值。

```
int i = 0, j = 0, k = 0;
const int ci = i; // 这两行都是初始化，而非赋值
1023 = k; // 错误，字面值是右值
i + j = k; // 错误， 算术表达式是右值
ci = k; // 错误， ci是不可修改的常量左值
```

2. 赋值运算对象的结果是他的左侧运算对象，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。**如果赋值运算符的左右两个运算对象类型不同，则右侧运算对象将转换成左侧运算对象的类型。** C++11新标准允许使用花括号括起来的初始值列表作为赋值语句的右侧运算对象。在这种情况下，**如果左值是一个内置类型，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（比如不能赋值给一个short类型的值超过short的存储的范围）。

```
k = {3.14}; // 错误，窄化转换
vector<int> vi;
vi = {0, 1, 2, 3}; // 正确，新标准允许这种赋值
```

3. 无论左侧运算对象的类型是什么，初始值列表都可以为空，此时，编译器创建一个值初始化的临时量并将其赋值给左侧对象。

4. **赋值运算符满足右结合律**：

```
int i, j;
i = j = 0; // 正确，i和j都被赋值为0。 0先被赋值给j，然后j赋值给i
int *ptr;
i = ptr = 0; // 错误，不能把int* 赋值给一个int对象

double d;
d = i = 3.5; // d是3.0, i是3
i = d = 3.5; // d是3.5，i是3
```

5. 对于多重赋值语句中的每一个对象，它的类型或者与**右边对象的类型相同**、或者**可由右边对象的类型转换得到**。

6. 赋值运算符的优先级低于关系运算符的优先级，所以在条件语句中，赋值部分通常应该加上括号：

```
int i;
while( (i = get_val()) != 31){
    // code block, 上方条件语句中如果不给赋值语句加上括号，
    // 那么将先判断get_val()和31的关系然后返回布尔值赋值给i
}
```

7. **切勿混淆相等运算符和赋值运算符！** if(i == j) 和 if(i = j)不同，后者是将j赋值给i然后转换成布尔值来进行判断

8. 任何一种复合运算符(+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)都完全等价于 a = a *op* b。两者的唯一区别在于运算的次数，复合型只运算了一次，而常规运算操作则运算了两次（第一次对右边式子求值，第二次是赋值给左侧的对象）

4.5 递增和递减运算符

1. 递增和递减运算符有两种形式：前置版本（++i）和后置版本（i++）。除非必须，否则不用递增递减的后置版本。原因在于前置版本把值加1后直接返回并改变了对象，而后置版本则需要先将原始值存储下来，然后进行加减操作，最后再返回之前存储的未修改的值，在这种情况下，如果我们不需要原始值，那么将对内存是一种浪费。

```
int i = 0, j;
j = ++i; // 先对i进行加操作，在赋值给j
j = i++; // 先保存i的原始值，再对i进行加操作，再把原始值赋值给j
```

2. 下面的代码中，++操作的优先级高于解引用（*），因此*pbeg++等价于*（pbeg++）。所以此处先对pbeg进行++操作，然后再返回其原始值给解引用操作符，此时解引用操作符返回的是pbeg未增加之前的值。此处可以有效避免解引用操作符操作一个先进行++再返回++之后的对象，因为此时对象可能已经指向一个未知的或不存在的元素，导致解引用发生不可预知的错误。

```
cout << *iter++ << endl; // 习惯这种操作，更加简洁也更少出错

cout << *iter << endl; // 尽量避免这种操作
iter++;
```

3. 因为递增运算符和递减运算符会改变运算对象的值，所以要提防在复合表达式中错用这两个运算符。

```
while(beg != s.end() && !isspace(*beg))
    *beg = toupper(*beg++); // 错误，赋值运算符两端的运算对象都用到了beg，
                            // 并且右侧的运算对象还改变了beg的值，此时无法知道先运算
                            // 所以该赋值语句是未定义的。

if(vec[i++] <= vec[i]){
    // 同样地，此时也不能明确左右哪边先运算，也会导致该赋值是未定义的
}
```

4.6 成员访问运算符

1. 解引用运算符的优先级低于点运算符，所以执行解引用运算的子表达式两段必须加上括号，如果不加则含义大不相同：

```
string s1 = "Hello", *p = &s1;
auto n = s1.size();
```

```
(*p).size(); // 先解引用p获取string对象，在调用size函数
*p.size(); // 错误：p是一个指针，没有size成员函数
```

2. 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，反之如果成员所述的对象是右值，那么结果是右值。

3. 4.20例题，假设iter的类型是vector::iterator,说明下面的表达式是否合法：

```
*iter++; // 解引用操作优先级小于++，所以先保存值再加加再返回原来的值，合法
(*iter)++; // 先解引用为string类型，再++，string++无意义，不合法
*iter.empty(); // 解引用操作优先级小于点操作，先对iter进行empty成员函数，指针无该成员
iter->empty(); // 合法
++*iter; // 先进行++，但++操作符操作于*iter解引用的string类型上，无意义，不合法
iter++->empty(); // ++操作符优先级低于箭头操作符，先调用empty成员函数，再进行++，合法
```

4.7 条件运算符

1. 条件运算符的形式：cond ? expr1 : expr2 其中cond是判断条件的表达式，而expr1和expr2是两个类型相同或可能转换成某个公共类型的表达式。执行过程是：先求cond的值，如果条件为真再求expr1的值并返回，否则对expr2进行求值。

```
string finalgrade = (grad < 60) ? "Fail" : "Pass";
```

2. 当条件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，否则运算结果为右值。

3. 条件运算符的内部允许嵌套另外一个条件运算符。

```
string finalgrade = (grad < 60) ? "Fail" : (grad > 90) ? "High Pass" : "Pass";
```

4. 条件运算符满足右结合律，意味着运算对象（一般）按照从右向左的顺序组合。因此上述代码中，靠右边的条件运算（grad > 90）构成了左边的条件运算的分支。

5. 随着条件运算嵌套层数的增加，代码可读性急剧下降。所以**最好不要超过三层嵌套**。

6. 条件运算符的优先级非常低，因此当一条长表达式中嵌套了条件运算子表达式时，通常需要再它两端加上括号。

```
cout << ((grade < 60) ? "fail" : "pass"); // 输出pass或者fail
cout << (grade < 60) ? "fail" : "pass"; // 输出0或者1,解释在下条
cout << grade < 60 ? "fail" : "pass"; // 错误，试图比较cout和60
```

7. 上述第二个式子，grade和60的比较结果是<<运算符的运算对象，因此如果grade < 60，则输出1，否则输出0，之后返回的仍然是cout对象，接下来cout作为条件运算符的条件。因此该式子等价于：

```
cout <<(grad <60); // 输出0或1
cout ? "fail" : "pass"; // 根据cout是true还是false产生相应的字面值
```

8. 第三个表达式则等价于：

```
cout << grade;
cout < 60 ? "fail" : "pass"; // 此处比较cout和60,此处为错误
```

4.8 位运算符

1. 位运算符作用于整数类型的运算对象，并把运算对象看做是二进制的集合。位运算符提供检查和设置二进制位的功能。标准库提供了一个名为bitset的标准库类型，可以表示任意大小的二进制位集合。所以位运算符同样能用于bitset类型。

2. 位运算符符合左结合律：

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

3. 一般来说，如果运算对象是“小整型”，则他的值会被自动提升成较大的整数类型。**注意：如果运算对象是带符号的且他的值是为负，那么位运算符号如何处理运算对象的“符号位”依赖于机器。而且，此时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。所以强烈建议仅将位运算符用于处理无符号类型。**

4. 移位运算符（<<和>>）的内置含义是对其运算对象执行基于二进制的位移动操作。首先令左侧运算对象的内容按照右侧运算对象的要求移动指定位数，然后将经过移动的左侧运算对象的拷贝作为求值结果。其中，**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**。二进制位或者向左移或者向右移，移出边界之外的位就会被舍弃掉了。

```
unsigned char bits = 0233; // 10011011
bits << 8; // bits提升成了int型，然后左移8位
        // 00000000 00000000 10011011 00000000
bits << 31; // 左移31位，左边超出边界的位丢弃掉了
```

```
            // 10000000 00000000 00000000 00000000
bits >> 3; // 向右移动3位，最右边的3位丢弃掉了
            // 00000000 00000000 00000000 00010011
```

5. **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运算对象的类型：如果是无符号类型，则在左侧的二进制位插入0，如果是带符号，则左侧插入符号位的副本或值为0的二进制数。具体视环境而定。**

6. 位求反(~)则是将运算对象逐位求反。

```
unsigned char bits = 0227; // 10010111
~bits; //先在前面加0提升为int型，再依次求反。
       // 00000000 00000000 00000000 10010111 提升为int
       // 11111111 11111111 11111111 01101000 依次求反
```

7. 位操作

   - '&'：通常将某些位清0且同时保持其他位不变

```
int n = 5;
n = n & 0xffffff00 //将n的低八位重置为0
```

   - '|': 通常将某些位变成1且同时保持其他位不变

```
int n = 5;
n = n | 0xffffff00 //将n的低八位重置为1
```

   - '^':异或操作，只有**相同时为0，否则为1**，用于变量中某些位的取反

```
int n = 5;
n = n ^ 0xff //将n的低八位取反
```

   - '<<'：左移运算符，高位丢弃低位补0.左移一位相当于乘以2，左移n位相当于乘以2^n。**左移方法比乘法操作更快！！**

   - '>>': 原符号为0则右移高位补0，原符号为1则右移高位补1。右移操作n位相当于除以2^n并将结果向小一个数取整。 例如：
     $-25 >> 4 = -25/2^4 = -2$
     $18 >> 4 = 18/2^4 = 1$

8. 异或操作的实际应用：数据交换。如果 a^b = c, 那么 b^c = a, a^c = b

```
int a = 1, b = 2;
a = a^b;
```

```
b = b^a;
a = a^b;// a和b的值互换
```

9. 位移操作符满足左结合律，它们的重载版本可以进行IO操作。位移操作的优先级不高不低，介于中间：比算术运算符的优先级低，但比关系运算符、赋值运算符和条件运算符的优先级高。因此在一次使用多个运算符时，有必要适当的地方加上括号使其满足我们的要求。

```
cout << 42 + 10; // 正确，+优先级更高，因此输出求和的结果
cout << (10 < 42); // 正确，括号使得运算对象按照我们的期望组合在一起，输出1
cout << 10 < 42; // 错误：试图比较cout对象和42的大小
```

10. 例题4.27：

```
unsigned long ull = 3, ul2 = 7;
ull & ul2; // 3
ull | ul2; // 7
ull && ul2; // 1
ul1 || ul2; // 1
```

4.9 sizeof运算符

1. sizeof运算符返回一条表达式或一个类型名字所占的字节数。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。

2. 运算符的运算对象有两种形式：

   - sizeof(type)
   - sizeof expr

3. 在上述第二种形式中，sizeof返回的是表达式结果类型的大小。与众不同的一点是，sizeof并不实际计算其运算对象的值：

```
Sales_data data, *p;
sizeof(Sales_data); // 存储Sales_data类型的对象所占的空间大小
sizeof data;// data的类型的大小，即sizeof(Sales_data)
sizeof p; //指针所占空间大小
sizeof *p; // p所指类型的空间大小，即sizeof(Sales_data)
sizeof data.revenue; // revenue成员所对应的大小
sizeof Sales_data::revenue; // 另一种获取revenue大小的方式
```

4. 上述例子中最有意思的是sizeof *p，首先因为sizeof满足右结合律并且与*的运算符的优先级一样，所以表达式按照从右向左的顺序组合。也就是说，他等价于sizeof(*p)。其次，因为sizeof不会实际求运算对象的值，所以即使p是一个无效的（未初始化的）指针，也不会有什么影响。在sizeof的运算对象中解引

用一个无效指针仍然是一种安全的行为，因为指针实际上并没有被真正的使用。sizeof不需要真的解引用指针也能知道他所指对象的类型。

5. c++11新标准允许我们使用作用域运算来获取类成员的大小。通常情况下只有通过类的对象才能访问到类的成员，但是**sizeof运算符无须我们提供一个具体的对象**，因为要想知道类成员的大小无须真的获取该成员。

    - 对char或者类型为char的表达式执行sizeof运算，结果为1
    - 对引用类型执行sizeof操作得到被引用对象所占空间的大小
    - 对指针执行sizeof运算得到指针本身所占空间的大小
    - 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，指针不需有效
    - 对数组执行sizeof运算得到整个数组所占空间的大小，等价于对数组中所有的元素各执行一次sizeof运算并将所得结果求和。注意，sizeof运算不会把数组转换成指针来处理。
    - 对string对象或vector对象执行sizeof操作，只返回该类型固定部分的大小，不会计算对象中元素占用了多少空间。

6. sizeof的返回值是一个常量表达式，所以我们可以用sizeof的结果声明数组的维度。

```
// 可以用数组的大小除以单个元素的大小得到数组中元素的个数
constexpr size_t sz = sizeof(ia) / sizeof(*ia);
int arr2[sz]; // 正确，sizeof返回的是一个常量表达式（数组在声明时长度必须是常量）
```

7. 例题4.29

```
int ar[10];
int *p = ar;
cout << sizeof(ar) / sizeof(*ar) << endl; // 10
cout << sizeof(p) / sizeof(*p) << endl;
// 返回1（32位系统）或者2（64位系统）,sizeof(p)返回指针大小
return 0;
```

4.10 逗号运算符

1. 和逻辑与、逻辑或以及条件运算符一样，逗号运算符也规定了运算对象求值的顺序。逗号运算符按照从左向右的顺序依次求值。

2. **逗号运算符首先对左侧的表达式求值，然后将求值结果丢弃掉**，逗号运算符真正的结果是右侧表达式的值。如果右侧运算对象的值是左值，那么最终的求值结果也是左值。

3. 习题4.33:

    someValue ? ++x, ++y : --x, --y 逗号表达式的优先级是最低的。因此这条表达式也等于：

    (someValue ? ++x, ++y : --x), --y 如果 someValue的值为真，x 和 y 的值都自增并返回 y 值，然后丢弃 y 值，y递减并返回 y 值。如果 someValue的值为假，x 递减并返回 x 值，然后丢弃 x 值，y递减并返回 y 值。

4.11 类型转换

1. 如果两种类型可以相互转换（conversion），那么他们就是关联的。
2. 当类型转换是自动进行的，无须程序员介入，有时甚至不需要程序员了解。他们被称作隐式转换。（implicit conversion）

```
int ival = 3.145 + 1; // 1先被转换成double，计算完后在转换成int为ival初始化
```

3. **算术类型之间的隐式转换被设计的尽可能避免损失精度**，很多时候，如果表达式中既有整数又有浮点数那么整型会被转换成浮点型。
4. 隐式转换的条件：
    - 比int类型小的整型值首先提升为较大的整数类型
    - 在条件中，非布尔型转换成布尔型
    - 初始化过程中，初始值转换成变量的类型，在赋值语句中，右侧运算对象被转换成左侧运算对象的类型。
    - 如果算术运算或关系运算的运算对象有多重类型，需要转换成同一种类型。
    - 函数调用时也会发生类型转换。

4.11.1 算术转换

1. 算术转换（arithmetic conversion）的含义是把一种算术类型转换成另一种算术类型。运算符的运算对象将转换成最宽的类型：
    - 如果一个运算对象是long double，那么所有的运算对象的类型都会转换成long double。
    - 当既有浮点也有整数时，整数类型都会转换成浮点型
2. 整型提升：对于bool、char、unsigned char、signed char、short和unsigned short等类型来说只要他们所有可能的值都能存在int里，那么他们就会被提升成int型。否则生成unsigned int。对于较大的(char, wchar_t, char16_t和char32_t)提升成int\unsigned int\long\unsigned long\long long和unsigned long long中最小的一种类型。前提是转换后的类型要能容纳元类型的所有可能的值。
3. 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，那么小类型的运算对象则会转换成较大的类型。
4. 如果一个运算对象是无符号类型，那么另一个运算对象是带符号类型，而且其中的无符号类型不小于带符号类型，那么带符号的运算对象转换成无符号的。如果有符号类型为负值，则结果可能带来各种副作用。
5. 如果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，则带符号类型的运算对象转换成无符号类型。例如一个long类型和一个unsigned int类型，且long和int都是32位，那么此时long就会转换成unsigned int类型，如果long比int占得空间更多，则unsigned int类型的运算对象转换成long类型。
6. 例子：

```
3.141412L + 'a'; // char转换成int,'a'对应97
char + float; // char转换成int再转换成float
short + char; // 两者都提升成int
char + long; // char转换成long
int + unsigned long; // int转换成unsigned long
unsigned short + int; // 根据两者所占空间进行提升
unsigned int + long; // 根据两者所占空间进行提升
```

4.11.2 其他隐式转换

1. 数组转换成指针：在大多数用到的数组的表达式中，数组自动转换成指向数组首元素的指针：

```
int ia[10];
int *p = ia; // ia自动转换成了指向首元素的指针
```

2. 当数组被用作decltype关键字的参数，或者作为取地址符(&)\sizeof\typeid等运算符的运算对象时，上述转换不会发生。如果用一个引用来初始化数组，上述转换也不会发生。

```
int (*PArray)[10] = &ia; // 取地址符不会使对象发生转换
int (&array)[10] = ia; // 引用初始化一个数组也不会发生转换
```

3. c++还规定了几种其他的指针转换方式，包括常量整数值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换成void*静态指针。指向任意对象的指针能转换成const void*。

4. 指针也可以转换成布尔类型，如果指针的值为0，则转换结果为false，否则为true。

5. 允许将指向非常量类型的指针转换成相应的常量类型的指针。对于引用也是这样。也就是说，如果T是一种类型，那么我们就能将指向T的指针或引用分别转换成指向const T的指针或const 引用。相反则不存在，因为他们尝试在删除底层const。

```
int i;
const int &j = i; // 非常量转换成const int的引用
const int *p = &i; // 非常量的地址转换成const的地址
int &r = j, *q = p; // 错误，不允许const转换成非常量
```

6. 类类型定义的转换由编译器自动执行的转换，不过编译器每次只能执行一种类类型的转换，如果同时提出多个转换请求，这些请求将被拒绝：

```
string s, t = "a value"; // 字符串字面值转换成string类型
while(cin >> s){ // while的条件部分把cin转换成布尔值

    // code block
}
```

4.11.3 显示转换

1. 强制转换类型（cast）本质上非常危险。

```
int i, j;
double slope = i/j; // 如果想使用浮点数除法，则必须得把i或j显式的转换成double
```

2. 显示转换的形式：cast-name(expression)，其中type是转换的目标类型，expression是要转换的值，如果type是引用类型，则结果是左值，cast-name是static_cast\dynamic_cast\const_cast\reinterpret_cast的一种。dynamic_cast支持运行时类型识别。

3. static_cast:任何具有明确定义的类型转换，只要不包含底层const，都可以使用static_cast。例如，通过将一个运算对象强制转换成double类型以执行浮点数除法：

```
double slope = static_cast<double>(j) / i;
```

4. 当需要把一个较大的算术类型赋值给较小的类型时，static_cast非常有用。此时强制类型转换会告诉读者和编译器，我们知道并且不在乎潜在的精度损失。（如果编译器发现一个较大的算术类型试图赋值给较小的类型，就会给出警告信息，但是当执行了显式的类型转换后，警告信息就会被关闭了）

5. static_cast可以找回存在于void*指针中的值，但要确保强制转换回来时指针的值保持不变，即强制转换的结果将与原始的地址相等，也就是必须确保转换后所得的类型就是指针所指的类型：

```
double d = 1.2321;
void* p = &d; // 任何非常量对象的地址都能存入void\*
double *dp = static_cast<double *>(p); // 将void\*转换回初始的指针类型
int* dp3 = static_cast<int *>(d); // 危险的操作，此时结果是未定义的
```

6. const_cast只能改变运算对象底层const

```
const char *pc;
char *p = const_cast<char*>(pc); //正确
// 如果对象本身不是一个常量，使用强制类型转换获得写权限是合法的，
// 如果对象是常量，那么再使用const_cast执行写操作就会产生未定义的后果
```

7. const_cast只能改变表达式的常量属性，不能改变一般表达式的类型：

```
const char *cp;
static_cast<char *>(cp); // 错误，static_cast不能换掉const性质
static_cast<string>(cp); // 正确，字符串字面值换成string类型
const_cast<string>(cp); // 错误，const_cast只能改变常量属性，不能改变一般表达式的类型
```

8. reinterpret_cast通常为运算对象的位模式提供较低层次上的重新解释。使用reinterpret_cast是非常危险的。该种转换本质上依赖机器。**要想安全地使用reinterpret_cast必须对设计的类型和编译器实现转换的过程都非常了解。**

```
int a = 10;
int *ip = &a;
char *pc = reinterpret_cast<char *>(ip); // 没有定义如何将int *转换为char *，此时只
                                         // 然而pc所指的真实对象仍然是一个int型对象
                                         // 来使用就可能在运行时发生错误
```

9. **建议：尽量避免使用强制类型转换。在有重载函数的上下文中使用const_cast无可厚非，但是其他情况下使用const_cast也就意味着程序存在某种设计缺陷。其他强制类型转换，比如static_cast和dynamic_cast都不应该频繁使用。每次书写了一条强制类型转换语句，都应该反复斟酌能否以其他方式实现相同的目标。就算实在无法避免，也应该尽量限制类型转换值得作用域，并记录相关类型的所有假定，这样可以减少错误发生的机会。**

10. 旧式的强制转换类型：

    - type (expr); // 函数形式的强制类型转换
    - (type) expr; // c语言风格的强制类型转换

```
int a = 10;
int *ip = &a;
char *pc = (char*) ip; // 效果与reinterpret_cast一样
```

11. 旧式的强制类型转换从表现形式上来看并不清晰，容易被看漏，所以一旦转换过程出现问题，追踪起来也更加困难。

4.12 c++运算符优先表

1. c++运算符优先表：

优先级	运算符	描述	例子	结合性
1	::	域运算符符	Class::age = 2;	
2	()	圆括号	(a + b) / 4;	从左到右
	[]	数组访问	array[4] = 2;	
	->	指针的成员访问	ptr->age = 34;	
	.	对象的成员访问	obj.age = 34;	
	++	后++	for(i = 0; i < 10; i++) ...	
	--	后--	for(i = 10; i > 0; i--) ...	
3	!	逻辑非	if(!done) ...	
	~	按位取反	flags = ~flags;	
	++	前++	for(i = 0; i < 10; ++i) ...	
	--	前--	for(i = 10; i > 0; --i) ...	
	-	负号	int i = -1;	从右到左
	+	正号	int i = +1;	
	*	取值运算符	data = *ptr;	
	&	取地址运算符	address = &obj;	
	(type)	强制类型转换	int i = (int) floatNum;	
	sizeof	字节大小运算符	int size = sizeof(floatNum);	
4	->*	成员指针运算符	ptr->*var = 24;	从左到右
	.*	成员对象运算符	obj.*var = 24;	
5	*	乘	int i = 2 * 4;	
	/	除	float f = 10 / 3;	从左到右
	%	余数（取模）	int rem = 4 % 3;	
6	+	加	int i = 2 + 3;	从左到右
	-	减	int i = 5 - 1;	
7	<<	左移	int flags = 33 << 1;	从左到右
	>>	右移	int flags = 33 >> 1;	
8	<	小于	if(i < 42) ...	
	<=	小于等于	if(i <= 42) ...	从左到右
	>	大于	if(i > 42) ...	
	>=	大于等于	if(i >= 42) ...	
9	==	等于	if(i == 42) ...	从左到右
	!=	不等于	if(i != 42) ...	
10	&	按位与	flags = flags & 42;	从左到右
11	^	按位异或	flags = flags ^ 42;	从左到右
12	|	按位或	flags = flags | 42;	从左到右
13	&&	逻辑与	if(conditionA && conditionB) ...	从左到右
14	||	逻辑或	if(conditionA || conditionB) ...	从左到右
15	? :	条件运算符	int i = (a > b) ? a : b;	从右到左
16	=	赋值运算符	int a = b;	
	+=	加后赋值	a += 3;	
	-=	减后赋值	b -= 4;	
	*=	乘后赋值	a *= 5;	
	/=	除后赋值	a /= 2;	
	%=	取模后赋值	a %= 3;	从右到左
	&=	按位与后赋值	flags &= new_flags;	
	^=	按位异或后赋值	flags ^= new_flags;	
	|=	按位或后赋值	flags |= new_flags;	
	<<=	左移后赋值	flags <<= 2;	
	>>=	右移后赋值	flags >>= 2;	
17	throw	抛出异常	throw EClass("Message");	从左到右
18	,	逗号运算符	for(i = 0, j = 0; i < 10; i++, j++) ...	从左到右