第4章 表达式

4.1 基础

4.1.1 基本概念

1. 当一个对象被用作右值时的时用的是对象的值（内容），当对象被用作左值的时用的是对象的身份（在内存中的位置）。

4.1.3 求值顺序

1. 运算对象的求值顺序与优先级和结合律无关。
2. 如果改变了某运算对象的值，在表达式的其他地方不要再使用这个运算对象。（当改变运算对象的子表达式本身就是另外一个子表达式的运算对象时该规则无效）

4.2 算术运算符

1. 算术运算符的运算对象和求值结果都是右值。
2. 在取余运算中， *m* 和 *n* 是整数且 *n* 非 *0*，除了 *-m* 导致溢出的特殊情况，其他时候 *m%(-n)* 等于 *m%n*, *(-m)%n* 等于 *-(m%n)*。

4.4 赋值运算符

```
// 1. 错误：窄化转换
int k = 0;
k = {3.14};
```

2. 因为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括号。

4.5 递增和递减运算符

1. 这两种运算符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。
2. 使用后置版本，对于整数和指针类型来说，编译器可能对这种额外的工作进行一定的优化。但是对于复杂的迭代器类型，这种额外的工作就消耗巨大。建议养成使用前置版本的习惯。
3. 后置运算符的优先级高于解引用运算符，因此 `*pbeg++` 等价于 `*(pbeg++)` 。
4. 大多数运算符都没有规定运算对象的求值顺序。下列代码将产生未定义的行为：

```
while (beg != s.end())
    *beg = toupper(*beg++);     // 错误：该赋值语句未定义

// 编译器可能按照下面的任意一种思路处理该表达式：
*beg = toupper(*beg);           // 如果先求左侧的值
*(beg + 1) = toupper(*beg);     // 如果先求右侧的值
```

4.6 成员访问运算符

1. 由于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端必须加上括号。

```
ptr->mem;
(*ptr).mem; // 等价
*ptr.mem;   // 不等价
```

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。
3. 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，如果成员的对象是右值，那么结果是右值。

4.7 条件运算符

1. 当条件运算符 `(cond ? expr1 : expr2)` 的两个表达式都是左值或者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。

```
int a = 0, b = 0, c = 1;
bool d = false;

d? a : b = c;        // 运算结果是左值
c = d? 0 : 1;        // 运算结果是右值
```

2. 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于表达式时，通常需要在它两端加上括号。

```
// 正确表达，输出 pass 或者 fail
cout << ((grade < 60) ? "fail" : "pass");


// 错误表达，输出 1 或者 0!
cout << (grade < 60) ? "fail" : "pass";
// 其表达等价于：
cout << (grade < 60);        // 输出 1 或 0
cout ? "fail" : "pass";      // 根据 cout 的值的真假产生对应的字面值


// 错误表达，试图比较 cout 和 60
cout << grade < 60 ? "fail" : "pass";
// 其表达等价于
cout << grade;      // 小于运算符的优先级低于位移运算符，所以先输出grade
cout < 60 ? "fail" : "pass";    // 比较 cout 和 60!
```

4.8 位运算符

1. 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。
2. 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。
3. 关于符号位如何处理没有明确规定，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 `>>` 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，否则会发生未定义行为。
5. 移位运算符的优先级不高不低，介于中间：比算术运算符的优先级低，但是比关系运算符、赋值运算符和条件运算符的优先级高。

4.9 sizeof 运算符

1. 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使用。 sizeof 不需要真的解引用指针也能知道它所指对象的类型。
2. 对 char 或者类型为 char 的表达式执行 sizeof 运算，结果为1。
3. 对数组执行 sizeof 运算得到整个数组所占空间的大小，该运算不会把数组转换成指针来处理。
4. 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，不会计算对象的元素占用了多少空间。

4.10 逗号运算符

1. 对于逗号运算符来说，首先对左侧的表达式求值，然后将求值结果丢弃掉。运算符真正的结果是右侧表达式的值。

4.11 类型运算符

4.11.1 算术转换

1. 某个运算符的运算对象类型不一致，一个对象是无符号类型，一个是有符号类型，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。
2. 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为带符号类型。如果不能，那么带符号类型的运算对象转换成无符号类型。

4.11.3 显示转换

1. 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 。可以使用 static_cast 找回存在于 void* 指针中的值。
2. const_cast： 只能改变运算对象的底层 const 。
3. 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：
   (1) 使用 reinterpret_cast 是非常危险的，其中的关键问题是类型改变了，但是编译器却没有给出任何警告或者错误的提示信息。
   (2) reinterpret_cast 本质上依赖于机器。想要安全的使用必须对涉及的类型和编译器实现转换的过程都非常熟悉。

4. 强制类型转换干扰了正常的类型检查，强烈建议程序员避免使用。实在无法避免的时候应该尽量限制类型转换值的作用域，并且记录对相关类型的所以假定，这样可以减少错误发生的机会。
5. 旧式的强制类型转换：

```
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
```