第四章 表达式

基础

- 把一个运算符和一个或多个运算对象组合起来可以生成较复杂的表达式

> 单个变量是表达式吗？ 如果是 那么 decltype(a) 为什们返回的是一个int 而不是 int&

- 函数调用也是一种表达式，它对运算的对象数量没有限制
- **重载运算符**：当运算符作用于类类型的运算对象时，用户可以自定义其含义

> IO库的 `>>` 和 `<<` 运算符，以及 `string` 对象和迭代器使用的运算符都是重载的运算符

- **左值和右值**：
  - 左值代表一个在内存中占有确定位置的对象（换句话说就是有一个地址）；rvalue是在不在内存中占有确定位置的表达式、
  - 左值使用的是对象的身份（在内存中的位置）；

    右值使用的是对象的值（内容）
  - 左值可以转换为右值

> **如果一个对象/表达式 既可以做左值，又可以做右值，那么它大概率是一个左值，比如说 *(p+1) 可以做左值也可以做右值，所以它是一个左值；**

```
左值：
    赋值运算符得到的结果
    解引用运算符、下标运算符返回的结果
    自增自减运算符返回的结果

右值：
    取地址运算符作用一个左值，但返回的指针是一个右值
    算术运算符的结果
```

- **复合表达式中求值顺序**：尽管复合表达式中的运算符有优先级，规定了运算对象的组合方式，但没有说明运算的对象按什么顺序执行；

```
int i = f() + g() *  h() + g();
/*
优先级规定，先做乘法再做加法
结合律规定，f先和乘法加，加完的结果再和g加
但是，对这些函数的调用顺序没有明确规定
*/
```

> 如果 函数/表达式 修改了同一对象，那么将会引发错误并产生未定义的行为

运算符

算术运算符

- 算术运算符都能作用于任意算术类型[1]
- 所有的运算对象都会转换为同意类型进行运算
- **C++11** 规定商一律向0取整

逻辑和关系运算符

- 关系运算符作用于算术类型或指针类型：

  逻辑运算符作用于任意能转换成布尔值的类型（短路求值）
- 返回值都是布尔类型

```
if (i < j < k)      // 若k>1,则必为真。因为（i<j)返回布尔值最大也就是1
```

赋值运算符

- 赋值运算符左侧运算对象必须是一个可修改的左值

  > 不可修改的左值，是那些用 `const` 修饰的变量

- 赋值运算符满足右结合律；

  多重赋值的情况下，或者与右边对象类型相同或者可以由右边对象的类型转换得到

```
int ival,*pval;
ival = pval = 0;        // 错误 pval=0没错，但是ival = pval有问题，int* 和 int 不能互相转换
```

递增和递减运算符

- 可额外应用于迭代器

  > 因为许多迭代器本身不支持算术运算

- 两种形式：
  - 前置版本
  - 后置版本

  > **除非必须，否则不用后置版本，推荐使用前置版本**
  >
  > 后置版本需要将原始值存储下来，以便返回这个未修改的内容；而前置则避免不必要的工作

- 递增和递减运算符的优先级高于解引用运算符

```
cout<<*it++;    // 相当于 cout<<*it; ++iter; 这样写更简洁
```

- 避免在一个表达式中，同时改变同一个指（运算的顺序是不确定的）

成员访问运算符

点运算符 `.`：对象是一个类型

箭头运算符 `->`：对象是一个指针

```
ptr->mem 等价于 （*ptr).mem;
------------------------------
string s1 = "a string", *p = &s1;
int n = (*p).size();
n = p -> size();            // n值不变
```

- 解引用运算符的优先级低于点运算符

```
*p.size();      // 错误 相当于*(p.size())
```

条件运算符

- 条件运算符 `?:`：允许我们把简单的 `if-else` 逻辑嵌入到单个表达式中

```
cond?expr1:expr2;
```

  - `cond` ——判断条件的表达式. 如果为真，则返回 `expr1` ,否则返回 `expr2`
- 嵌套的条件运算符：允许在条件运算符内部嵌套另一个条件运算符

```
finalgrade = (grade > 90)?"high":(grade<60)?"fail":"pass";
```

  - 条件运算符满足右结合律；
  - 为了代码的可阅读性，嵌套最好别超过3层
- 条件运算符的优先级非常低，甚至比输出运算符都低

```
cout<<(grade<60)?"fail":"success";
等于
    cout<<(grade<60);
    cout ? "fail":"suc";    //cout 是一个ostream对象
```

位运算符

位运算符作用于**整数类型**的运算对象，并把对象看成二进制位的集合

- 位运算符

```
~expr               按位求反
expr1 << expr2      expr1 左移 expr2位
expr1 >> expr2      expr1 右移 expr2位
&                   位与
^                   位异或
|                   位或
```

> 关于符号位如何处理没有明确的规定，**强烈建议仅将位运算用于处理无符号类型**

- `<< >>`运算符的内置含义是其运算对象执行基于二进制位的移动操作（输出输出是其重载版本）；
  - 右侧对象expr2不能为负；
  - 移出边界之外的位被舍弃；
  - 左移在右侧补二进制0，右移则在左侧补最初最高位的二进制（可以理解为补符号位）

> 移位运算符满足左结合律；
>
> ```
> cout<<"hi"<"there"<<endl;
> (cout<<"hi")<<"there")<<endl;
> ```
>
> 优先级比算术运算符低，比关系运算符、赋值运算符和条件运算符高

```
cout<<1+2;
cout<<(10<43);
cout<<10<43;        // 错误 等于 （cout<<10)<43 试图比较 cout和43
```

sizeof运算符

`sizeof()` 返回一条表达式或一个类型名字所占的字节数；满足右结合律，所得值是一个 `size_t` 类型的常量表达式

- 两种形式：

```
sizeof(expr);
sizeof expr;
```

> 不会真正计算其运算对象的值
>
> ```
> sizeof *p;
> /*
> sizeof满足右结合律 且 与 *运算符优先级一样  等价于 sizeof(*p);
> 因为sizeof不会实际求运算对象的值，所以即使p是一个无效（即未初始化）的指针也不会有什么影响
> */
> ```

- 一些特殊情况
  - 对数组名执行，不会把数组转换成指针来处理，会返回数组内所有元素的sizeof之和
  - 对 `string对象` 或 `vector` 对象执行，只会返回该类型固定部分的大小，不会计算对象中的元素到底占用了多少空间

    > string实例变量里面存放的只是一个内存块的指针，新增的元素都以指针的形式相连，但不属于string内部元素

逗号运算符

含有两个运算对象，按照从左到右的顺序依次求值。

- 真正返回的结果是右侧表达式的值；

```cpp
int a,b;
int c;
c = (a=1,2);
// c = a=1,2;   c= 1
cout<<a<<" ";
cout<<c;    // c =2
```

- **逗号运算符的优先级最低**

类型转换

如果两种类型相关联，那么就可以相互转换

隐式转换：尽可能避免损失精度

算术转换规则

- 整型提升：小整数类型转换为较大整数类型
- 无符号 -> 有符号

数组转换指针

- 大多数用到数组的表达式中，数组自动转换为指向数组首元素的指针

> **特殊情况**：
>
> 数组被用作 `decltype` 的参数，作为取地址符 `&` 、 `sizeof` 和 `typeid` 等运算符的运算对象时，上述转换不会发生（即把数组名当成一个数组）

指针的转换

- 规定：
  - 常量整数值0 ， 或者字面值 nullptr能转换为任意指针类型；
  - 指向任意非常量的指针能转换为 `void*`
  - 指向任意对象的指针能转换为 `const void*`

> `void*` 到底有什么用?
>
> 应该是你想用指针，但是又还不知道他的类型。有点像多态的感觉。void *负责透传，传入方和使用方自己控制类型

转换为布尔类型

- 存在一种从算术类型或指针类型向布尔类型自动转换的机制

转换为常量

- 允许将指向非常量类型的指针转换成指向相应的常量类型的指针，对于引用也是这样

> 相反的转换并不存在，因为会试图删掉底层const `??`、

> 非常量类型是否一定可以转换为对应的常量类型?

类类型定义的转换

- 类类型能定义有编译器自动执行的转换，不过编译器每次只能执行一种隐式类类型的转换
- 例子：

```
string  s = "value";        // 字符串字面值转换为string类型
```

显示转换

> 尽量避免强制类型转换

命名的强制类型转换

```
cast-name<type>(expression)
```

- cast-name：指定指向的是哪种转换
  - `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

    > - 当需要将一个较大的类型转换为较小的类型时，使用该声明，编译器不会发出警告（这种声明用的较多）
    > - static_cast 不能用于在不同类型的指针之间互相转换，也不能用于整型和指针之间的互相转换，当然也不能用于不同类型的引用之间的转换

  - `const_cast`： **只能**改变运算对象的底层const

    ```
    const char *pc;     // const 限定的是*p
    char *p = const_cast<char*>(pc);    //正确；但是通过p写值
    ```

    > 只有该声明能改变表达式的常量属性。
    >
    > 同样，不能用该声明改变表达式的类型。

  - `reinterpret_cast`：为运算对象的**位模式**提供较低层次上的**重新解释**

```cpp
int *ip;
char *pc = reinterpret_cast<char*>(ip);      // pc所指的真实对象是一个int而非字符，当时这样写会把它当作一个字符指针
string str(pc);      // 可能会导致异常的运行时行为
```

> 进行各种不同类型的**指针之间**、不同类型的**引用之间**以及**指针和能容纳指针的整数类型之间**的转换。转换时，**执行的是逐个比特复制的操作**

- `drnamic_cast`：支持运行时类型识别 **后续章节**

- type：转换的目标类型
- expression：要转换的值

旧式类型转换

```cpp
(type) expr;
```

> C++ 引入新的强制类型转换机制，主要是为了克服C语言强制类型转换的以下三个缺点。
>
> 1. 没有从形式上体现转换功能和风险的不同。例如，将 int 强制转换成 double 是没有风险的，而将常量指针转换成非常量指针，将基类指针转换成派生类指针都是高风险的，而且后两者带来的风险不同（即可能引发不同种类的错误），**C语言的强制类型转换形式对这些不同并不加以区分。**
> 2. 将多态基类指针转换成派生类指针时不检查安全性，即无法判断转换后的指针是否确实指向一个派生类对象。
> 3. **难以在程序中寻找到底什么地方进行了强制类型转换，不好追踪**

练习

4.17

前置递增运算符：将对象本身作为左值返回，即以及自增的值

后置递增运算符：将对象的原始值的副本作为右值返回

4.18

1. 基本数据类型都属于算术类型 ↩