表达式

- C++ Primer第五版
- 第4章

1

表达式（expression）基础：

- 表达式由一个或多个**运算对象**（operand）组成，对表达式求值将得到一个结果。
- 字面值和变量是最简单的表达式。
- 把**运算符**和**运算对象**结合起来可以生产较复杂的表达式。

运算符

- 一元运算符：作用域一个运算对象的运算符，如取地址符（&）和解引用符（*）
- 二元运算符：作用于两个运算对象的运算符，如相等（==）和乘法（*）
- 还有一个作用于三个运算对象的三元运算符

完全不相干

- 表达式 的计算结果，依赖运算符的**优先级**（precedence）、**结合律**（associativity）以及运算对象的**求值顺序**（order of evaluation）
  - 例：5 + 10 * 20 / 2中，*的运算对象可能是：
    - 10*20、15*20、15*20/10、10*20/10

- 运算符重载
  - 当运算符作用于类类型的运算对象时，用户可以自行定义其含义。
  - 例：IO库的>>和<<，string对象、vector对象和迭代器使用的运算符

运算对象的个数、运算符的优先级、结合律都是无法改变的。

- C++表达式要么是**左值**，要么是**右值**
  - C语言：左值可以位于赋值语句的左侧、右值不能。
  - C++语言中，要复杂得多
    - 右值：取不到地址的表达式
    - 左值：能取到地址的表达式
      - 常量对象为代表的左值不能作为赋值语句的左侧运算对象
      - 某些表达式的求值结果是对象，但他们是右值

- 当一个对象被作用右值的时候，用的是对象的值（内存中的内容）
- 当一个对象被当做左值的时候，用的是对象的身份（内存中的位置）

通常情况：
- 左值可以当成右值，实际使用的是它的内容（值）
- 不能把右值当成左值（也就是位置）

- 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。
- 内置解引用运算符、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。

- 如果表达式的求值结果是左值，decltype作用于该表达式（不是变量）得到一个引用类型。例如，对于int *p ：
  - 因为解引用运算符生成左值，所有decltype(*p)的结果是int&
  - 因为取地址运算符生成右值，所以decltype(&p)的结果是int **

decltype与表达式

- 表达式 的计算结果，依赖运算符的优先级（ precedence ）、结合律（ associativity ）以及运算对象的求值顺序（ order of evaluation ）
  - 根据运算符的优先级：表达式 3 + 4*5 的值是23，不是35
  - 根据运算符的结合律：表达式 20-15-3的值是2，不是8

6 + 3 * 4 / 2 + 2 等价于 ((6+(3*4)/2)+2)

- 大多数情况下，不会明确指定求值顺序
  - int i = f1() * f2()；
    - f1和f2将在乘法之前被调用，但是不知道谁前谁后
  - 对于没有指定求值顺序的运算符来说，如果表达式修改了同一个对象，将会引发错误并产生未定义行为。
    - int i = 0; cout << i << “ “ << ++i <<endl; //未定义的

如果改变了某个运算对象的值，在表达式的其他地方不要再使用这个运算对象。

算术运算符：

算术运算符（左结合律），按优先级分组		
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

```
//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; //true
```

```
//运算符%俗称“取余”或“取模”运算符，计算两个整数相除所得的余数
int ival = 42;
double dval = 3.14;
ival % 12; //正确：运行结果是6
ival % dval; //错误
```

如果m和n死整数且非0：
表达式（m/n）*n+m%n的求值结果与m相等

```
//除非-m导致溢出：
//(-m)/n和m/(-n)都等于-(m/n)
//m%(-n)等于m%n,(-m)%n等于-(m%n)
21%6;  /*结果是3*/21/6;  /*结果是3*/
21%7;  /*结果是0*/21/7;  /*结果是3*/
-21%-8;  /*结果是-5*/-21/-8;  /*结果是2*/
21%-5;  /*结果是1*/21/-5;  /*结果是-4*/
```

逻辑和关系运算符：

逻辑运算符和关系运算符，按优先级分组

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

```cpp
//s是对常量的引用：元素即没有被拷贝也不会被改变
string text[4] = {"Hello"," ","","the world."};
for (const auto& s : text) {
	cout << s; //输出当前元素
	//遇到空字符或者以句号结束的字符串进行换行
	if (s.empty() || s[s.size() - 1] == '.')
		cout << endl;
	else
		cout << " "; //否则用空格隔开
}
```

```
Microsoft Visual Studio 调试控制台
Hello
the world.
```

```cpp
float i = 1, j = 0, k = 0.5;
//i<j的布尔值结果和k比较
if (i < j < k) //若k大于1，则一定为true（1）
		cout << "true" << endl;
else
		cout << "false" << endl;

if (i < j && j < k)
		cout << "true" << endl;
else
		cout << "false" << endl;
```

```
Microsoft Visual Studio 调试控制台
true
false
```

赋值运算符：

```
int i = 0, j = 0, k = 0; //初始化而非赋值
const int ci = i; //初始而非赋值

1024 = k; //错误
i +  j =  k;  //错误：算术表达式是右值
ci = k; //错误

k = 0;
k = 3.14159; //结果为3

//C++11允许使用初始化列表作为赋值语句右侧的运算对象
k =  {  3.14  };  //错误：窄化转换
vector<int> vi; //初始为空
vi = { 0,1,2,3,4,5,6,7,8,9 };
```

赋值运算满足右结合律

```
int ival, jval;
ival = jval = 0; //正确：都被赋值为0

int* pval;
ival =  pval =  0;  //错误：不能把指针赋值给int
string s1, s2;
s1 = s2 = "OK"; //字符串字面值"OK"转换成string对象
```

赋值运算优先级较低

```
//不断循环，直到遇到42
int i = get_value();
while (i!=42){
      //其他处理...
      i = get_value(); //得到下一个值
}

//更好的写法，条件部分表达跟清晰
int i;
while ((i = get_value()) != 42) {
            //其他处理...
}
```

递增和递减运算符：

```
int i = 0, j;
j = ++i; // j=1,i=1:前置版本，得到递增之后的值
j = i++; // j=1,i=2;后置版本，得到递增之前的值
```

```
auto pbeg = v.begin();
//输出元素直到遇到第一个负值
while (pbeg != v.end() && *beg >= 0)
        cout <<  *pbeg++  <<  endl;  //输出当前值并将pbg移动到下一个元素
```

注意：++的优先级高于*

运算对象可按任意顺序求值

```
//该循环的行为未定义
while (beg != s.end() && !isspace(*beg))
        * beg = toupper(*beg++);

//编译器可能按照下面的任意一种思路处理：
//先求左侧：*beg = toupper(*beg);
//先求右侧：*(beg+1) = toupper(*beg);
//也可能采用别的方法处理
```

成员访问运算符：

- **点运算符和箭头运算符都可以访问成员**
- **ptr->mem等价于(*ptr).mem**

```
string s1 = "a string", * 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();
```

条件运算符：cond ? expr1 : expr2

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

//允许嵌套（右结合律）
finalgrade = （grade > 90) ? "high pass"
                           : (grade < 60) ? "fail" : "pass";

cout << ((grade < 60) ? "fail" : "pass"); //输出fail或pass
cout << (grade < 60) ? "fail" : "pass"; //输出1或0
cout << grade < 60 ? "fail" : "pass"; //错误:试图比较cout和60
```

```
cout << (grade < 60);
cout ? "fail" : "pass";
```

```
cout << grade;
cout < 60 ? "fail" : "pass";
```

位运算符：

- 作用于整数类型的运算对象，把运算对象看成是二进制位的集合
- 检查和设置二进制位

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

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

```
//假设char占8位，int占32位
unsigned char bits = 0233; //八进制，二进制为：10011011
bits << 8; //bits提升为int型，然后向左移动8位

bits = 0227; //10010111
~bits;

unsigned char b1 = 0145;   //01100101
unsigned char b2 = 0257;   //10101111
b1 &  b2;  //24个高位都是0, 00100101
b1 | b2; //24个高位都是0，11101111
b1 ^  b2;  //24个高位都是0，11001010
```

如果运算对象是“小整形”、则它的值会被自动提升

使用位运算符：假设班级中有30个学生，用一个二进制位表示某个学生在测试中是否通过。

```
unsigned long quizl = 0; //把这个值当做位的集合，在任何机器都至少32位
quizl |=  1UL  <<  27;  //学生27通过了测试
quizl &=  ~(1UL  <<  27);  //学生27没有通过测试

bool status = quizl & (1UL << 27); //学生27是否通过了测试
```

sizeof运算符：

- 返回所占的字节数

```
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的类型大小，即sizeof(Sales_data)
sizeof p; //指针所占的空间大小

//sizeof和*优先级一样，并且满足右结合律
//所以下面等价于sizeof (*p)
//p可以是无效指针，并不会执行解引用
sizeof *p; //p所指类型的空间大小，即sizeof(Sales_data)

sizeof data.revenue; //Sales_data的revenue成员对应类型的大小
sizeof Sales_data::revenue; //另一种获取revenue大小的方式

//sizeof运算能够得到整个数组的大小
constexpr size_t sz = sizeof(ia) / sizeof(*ia);
int arr2[sz]; //正确：sizeof返回一个常量表达式
```

逗号运算符：

- 含有两个运算对象，按照从左向右的顺序依次求值
- 返回结果为右侧表达式的值

```
//逗号运算符经常被用在for循环当中
vector<int> ivec = { 1,2,3,4 };
vector<int>::size_type cnt = ivec.size();
//将把从size到1的值赋给ivec的元素
for (vector<int>::size_type ix = 0; ix != ivec.size(); ++ix, --cnt)
                    ivec[ix] = cnt;
```

类型转换：

- 隐式转换：无需程序员介入，程序员甚至无需了解。

```
int  ival =  3.541  +  3;  //为了避免损失精度,编译器先执行3.541+3.0
```

- 算术转换：理解算术转换，办法之一就是研究大量的例子

```
bool      flag;          char              cval;
short     sval;          unsigned short    unsval;
int       ival;          unsigned int      uival;
long      lval;          unsigned long     ulval;
float     fval;          double            dval;

3.14159L + 'a';     //'a'提升成int，然后该int值转换成long double
dval + ival;        //ival转换成double
dval + fval;        //fval转换成double
ival = dval;        //dval转换成（切除小数部分后）int
flag = dval;        //如果dval是0，则flag是false，否则flag是true
cval + fval;        //cval提升成int，然后该int值转换成float
sval + cval;        //sval和cval都提升成int
cval + lval;        //cval转换成long
ival + ulval;       //ival转换成unsigned long
unsval + ival;      //根据unsigend short和int所占空间的大小进行提升
uival + lval;       //根据unsigned int和long所占空间的大小进行转换
```

- 其他隐式类型转换

/*指针的转换*/
- 0或字面值nullptr能够转换成任意指针类型
- 指向任意非常量的指针能够转换成void*
- 指向任意对象的指针能够转换成const void*

```
/*数组转换成指针*/
int  ia[10];//含有10个整数的数组
int *ip = ia;//ia转换成指向数组首元素的指针

/*转换成布尔类型*/
char *cp = get_string();
if (cp) /*...*/ //如果指针cp不是0，条件为true
while (*cp) /*...*/ //如果*cp不是空字符，条件为true

/*转换成常量*/
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，因为试图删掉底层const

/*类类型定义的转换*/
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
```

- 显式转换：强制转换 cast-name<type> (expression)
  - cast-name是static_cast、dynamic_cast、const_cast和reinterpret_cast中的一

```
/*static_cast:只要不包含底层const，都可以使用*/
//进行强制类型转换以便指向浮点数除法
int i, j;
double slope = static_cast<double>(j) / i;

double d;
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

/*cosnt_cast:只能改变运算对象的底层const*/
const char *pc;
char *p = const_cast<char*>(pc); //正确：但通过p写值是未定义的行为

const char *cp;
char *q = static_cast<char *p>(cp); //static_cast不能转掉const性质
static_cast<string>(cp); //正确
const_cast<string>(cp); //错误

/*reinterpret_cast:为运算对象的位模式提供较低层次上的重新解释*/
int* ip;
char* pc = reinterpret_cast<char*>(ip); //效果类似C风格的强制转换
```

- 旧式的强制转换

```
char *pc = (char*) ip;//ip是指向整型的指针，在这里与reinterpret_cast一样
```