C++ 程序设计

第4章 表达式

内容

表达式

- 4.1 基础
- 4.2 算术运算符
- 4.3 关系和逻辑运算符
- 4.4 赋值运算符
- 4.5 自增和自减运算符
- 4.6 成员访问运算符
- 4.7 条件运算符
- 4.8 位运算符
- 4.9 sizeof 运算符
- 4.10 逗号运算符
- 4.11 类型转换
- 4.12 优先级表

表达式

- 表达式(expression)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(result)
- 单个字面值或变量是最简单的表达式
  - 其结果就说字面值或变量的值
- 把一个运算符(operator)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

基础

- 根据运算对象的个数，运算符可分为
  - 一元运算符，如取地址符（&）和解引用符（*）
  - 二元运算符，如相等运算符（==）和乘法运算符（*）
  - 三元运算符，条件运算符?:
  - 函数调用运算符，对运算对象的数目没有限制
- 一些符号既能作为一元运算符也能作为二元运算符，例如*

优先级与结合性

基础

- 复合表达式(compound expression)是指含有两个或多个运算符的表达式
- 求复合表达式的值需要首先将运算符和运算对象合理地组织在一起，而优先级(precedence)与结合性(associativity)决定了运算对象组合的方式
  - 高优先级运算符的运算对象要比低优先级运算符的运算对象更紧密地组合
  - 如果优先级相同，则组合规则由结合性来确定
  - 括号无视优先级与结合性

算术运算符

表 4.1：算术运算符（左结合律）

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

- 一元运算符优先级最高，接下来是乘法和除法，最低的是加法和减法
- 算术运算符都满足左结合性
- 在求值时，小整数类型的运算对象被提升成较大的整数类型，所有运算对象最终会转换成同一类型
  - 对大多数运算符，bool类型被提升为int型

+和-

算术运算

- 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（可能提升后的）副本
- 一元负号运算符对运算对象取负后，返回其（可能提升后的）副本

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

除法

算术运算

- 整数相除的结果还是整数
  - C++11标准规定商一律向0取值（即直接截断小数部分）

```
int ival1 = 21/6;  // ival1 is 3; result is truncated; remainder is discarded
int ival2 = 21/7;  // ival2 is 3; no remainder; result is an integral value
```

- 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;   // ok: result is 6
ival % dval; // error: floating-point operand
```

除法

算术运算

- 除法运算中，如果两个运算对象的符号相同则商为正，否则商为负
- 如果m和n是整数且m%n不等于0，则m%n的符号与m相同
  - (-m)/n和m/(-n)都等于-(m/n)
  - m%(-n)等于m%n，(-m)%n等于-(m%n)

```
21 % 6;   /*  result is 3  */      21 / 6;   /*  result is 3   */
21 % 7;   /*  result is 0  */      21 / 7;   /*  result is 3   */
-21 % -8; /*  result is  -5 */     -21 / -8; /*  result is 2   */
21 % -5;  /*  result is 1  */      21 / -5;  /*  result is  -4  */
```

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
  - 值为0的运算对象（算术类型或指针）表示false，否则为true
- 返回值都是bool类型，运算对象和求值结果都是右值
- 除了逻辑非是右结合，其他都满足左结合性

逻辑和关系运算符

表 4.2：逻辑运算符和关系运算符

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

逻辑运算符

逻辑和关系运算符

- 对逻辑与运算符来说，当且仅当两个运算对象求值为true时结果为true
- 对逻辑或运算符来说，只要两个运算对象中的一个求值为true时结果为true
- 逻辑非运算符将运算对象的值取反后返回
- 逻辑与、逻辑或运算符都是先求左侧运算对象的值再求右侧运算对象的值

关系运算符

逻辑和关系运算符

- 关系运算符（<, <=, >, <）比较运算对象的大小关系并返回布尔值
- 满足左结合性

```
// oops! this condition compares k to the bool result of i < j
if (i < j < k) // true if k is greater than 1!
```

➢正确的表达式为

```
// ok: condition is true if i is smaller than j and j is smaller than k
if (i < j && j < k) { /* ... */ }
```

赋值运算符

- 赋值运算符的左侧运算对象必须是一个可修改的值

```
int i = 0, j = 0, k = 0; // initializations, not assignment
const int ci = i;        // initialization, not assignment
1024 = k;      // error: literals are rvalues
i + j = k;     // error: arithmetic expressions are rvalues
ci = k;        // error: ci is a const (nonmodifiable) lvalue
```

赋值运算符

- 赋值运算的结果是它的左侧运算对象
- 结果的类型是左侧运算对象的类型
  - 如果左右运算对象类型不同，则右侧运算对象将转换为左侧运算对象的类型

```
k = 0;          //  result: type int, value 0
k = 3.14159;    //  result: type int, value 3
```

左结合性

赋值运算符

- 不同于其他二元运算符，赋值运算符具有右结合性

```
int ival, jval;
ival = jval = 0; // ok: each assigned 0
```

- 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

```
int ival, *pval; // ival is an int; pval is a pointer to int
ival = pval = 0; // error: cannot assign the value of a pointer to an int
```

较低优先级

赋值运算符

- 赋值运算符的优先级仅高于逗号运算符，所以通常给条件的赋值部分加上括号使其符合原意

```
// a verbose and therefore more error-prone way to write this loop
int i = get_value();  // get the first value
while (i != 42) {
    // do something ...
    i = get_value();  // get remaining values
}
```

改写为：

```
int i;
// a better way to write our loop---what the condition does is now clearer
while ((i = get_value()) != 42) {
    // do something ...
}
```

切勿混淆相等和赋值

赋值运算符

- 由于C++允许用赋值运算作为条件

  if (i = j)

- 初衷可能是想判断i和j是否相等

  if (i == j)

  - 程序的这种缺陷很难被发现，有些编译器会对类似的代码给出警告

复合赋值运算符

赋值运算符

- 算术运算符和位运算符都有相应的复合赋值形式：

```
 +=   -=   *=   /=   %=          // arithmetic operators
<<=  >>=   &=   ^=   |=          // bitwise operators
```

a *op*= b;

完全等价于

a = a *op* b;

练习

赋值运算符

- 练习4.13：当赋值完成后i和d的值分别是多少？

```
int i; double d;
(a) d = i = 3.5;
(b)  i = d = 3.5;
```

- 练习4.14：执行下述if语句后发生什么情况？

```
if (42 = i)   //  ...
if (i = 42)   //  ...
```

练习

赋值运算符

- 练习4.15：下列赋值为何非法，如何修改？

```
double dval; int ival; int *pi;
dval = ival = pi = 0;
```

- 练习4.16：尽管下面语句合法，但它们实际执行行为可能和预期不一样，为什么？如何修改？

```
(a) if (p = getPtr() != 0)
(b) if (i = 1024)
```

自增和自减运算符

- 自增(++)和自减(--)运算符有两种形式
  - 前置版本首先将运算对象加1（或减1），然后将改变后的对象作为求值结果
  - 后置版本也会将运算对象加1（或减1），但是求值结果是运算对象改变之前那个值的副本

```
int i = 0, j;
j = ++i; // j = 1, i = 1: prefix yields the incremented value
j = i++; // j = 1, i = 2: postfix yields the unincremented value
```

成员访问运算符

- 点运算符和箭头运算符都可用于访问成员
  - 点运算符获取类对象的一个成员
  - 箭头运算符与点运算符有关：*ptr->mem*等价于(**ptr*).*mem*
    - ✓解引用运算符的优先级低于点运算符，所以要加上括号

条件运算符

- 条件运算符（?:）允许把简单额if-else逻辑嵌入到单个表达式中

  cond ？*expr1* ：*expr2;*

- 执行过程：首先求cond的值，如果为真对expr1求值并返回该值，否则对expr2求值并返回该值

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

嵌套条件运算符

条件运算符

- 允许在条件运算符的内部嵌套另外一个条件运算符

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

- 条件运算符满足右结合性
  - 例子中靠右的条件运算构成了靠左的条件运算的：分支

输出表达式中使用条件运算符

条件运算符

- 条件运算符的优先级非常低，仅高于赋值运算符
  - 当长表达式中嵌套了条件运算子表达式时，通常需要在它两端加上括号

```
cout << ((grade < 60) ?   "fail" : "pass"); // prints pass or  fail
cout << (grade < 60) ?   "fail" : "pass";   // prints 1 or 0!
cout << grade < 60 ?   "fail" : "pass"; //  error: compares cout to 60
```

  - 第二条表达式等价于

```
cout << (grade < 60);    // prints 1 or 0
cout ?  "fail" : "pass"; // test cout and then yield one of the two literals depending on whether cout is true or false
```

  - 第三条表达式等价于

```
cout << grade;   // less-than has lower precedence than shift, so print grade first
cout < 60 ? "fail" : "pass"; // then compare cout to 60!
```

sizeof运算符

- sizeof运算符返回表达式或类型名所占的字节数，并不实际计算其运算对象的值
  - 有两种形式：

sizeof (*type*)

sizeof *expr*

  - 满足右结合性，运算结果是size_t类型的常量表达式

```
Sales_data data, *p;
sizeof(Sales_data); // size required to hold an object of type Sales_data
sizeof data; // size of data's type, i.e., sizeof(Sales_data)
sizeof p;    // size of a pointer
sizeof *p;   // size of the type to which p points, i.e., sizeof(Sales_data)
sizeof data.revenue; // size of the type of Sales_data's revenue member
sizeof Sales_data::revenue; // alternative way to get the size of revenue
```

sizeof运算符

- sizeof的运算结果部分地依赖于其作用的类型
  - 对char或char类型的表达式执行sizeof运算，结果是1
  - 对引用类型执行sizeof运算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。sizeof运算不会把数组转换为指针
  - 对string对象或vector对象执行sizeof只返回该类型固定部分的大小，不会返回对象的元素占用了多少空间

```
string b="hello";
vector<int> i={1,2,3};


cout<<sizeof b<< " " <<sizeof i<< endl;

// 24    12
```

逗号运算符

- 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
- 先对左侧表达式求值，然后将结果丢弃；逗号表达式的结果是右侧表达式的值

类型转换

- 下面表达式将ival初始化为6：

  int ival = 3.541 + 3; // *the compiler might warn about loss of precision*

- C++不会直接将两个不同类型的值相加，而是先根据类型转换规则设法将运算对象的类型统一后再求值

隐式类型转换

类型转换

- 在下面情况下，编译器自动进行隐式类型转换
  - 在大多数表达式中，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
  - 初始化时，初始值转换成变量的类型；赋值时，右侧运算对象转换成左侧运算对象的类型
  - 如果算术或关系表达式中的运算对象有多种类型，则转换成同一种类型
  - 函数调用时也会发生类型转换

算术转换

类型转换

- 整型提升(integral promotion)负责把小整数类型转换成较大的整数类型
  - 对于bool, char, signed char, unsigned char, short和unsigned short类型，只要它们所有可能的值能存在int里，它们就会提升成int类型；否则，提升成unsigned int类型
  - 较大的char类型（ wchar_t, char16_t和char32_t）提升成int, unsigned int, long, unsigned long, long long,或unsigned long long中最小的一种类型

无符号类型的运算对象

算术转换

- 如果某个运算对象是无符号类型，那么转换结果依赖于机器中各种整数类型的相对大小
  - 首先进行整型提升。如果结果的类型匹配，无须进行进一步的转换
  - 如果两个（可能提升的）运算对象有相同的符号，则小类型的运算对象转换成较大的类型
  - 如果符号不同，且无符号运算对象的类型不小于带符号类型，则带符号运算对象转换成无符号的
    - ✓例如，假设两个类型分别是unsigned int和int，则int类型的运算对象转换成unsigned int类型

无符号类型的运算对象

算术转换

- 如果带符号类型大于无符号类型，转换结果依赖于机器
  - ✓如果无符号类型的所有值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，如果两个运算对象分别是long和unsigned int类型，并且int和long的大小相同，则long类型的运算对象转换成unsigned int类型；如果long类型占用的空间多，则unsigned int类型的运算对象转换成long类型

显式转换

类型转换

- 如果想在下面的代码中执行浮点数除法

```
int i, j;
double slope = i/j;
```

- 就要使用某种方法将i和/或j显式地转换成double

命名的强制类型转换

显式转换

- 命名的强制类型转换(named cast)具有如下形式：

  cast-name<type>(expression);

  - type是转换的目标类型，而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和reinterpret_cast中的一种

static_cast

命名的强制类型转换

- 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

```
// cast used to force floating-point division
double slope = static_cast<double>(j) / i;
```

  - 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
  - static_cast对于编译器无法自动执行的类型转换也非常有用

```
void* p = &d;   // ok: address of any nonconst object can be stored in a void*
// ok: converts void* back to the original pointer type
double *dp = static_cast<double*>(p);
```

const_cast

命名的强制类型转换

- const_cast只改变运算对象底层const

```
const char *pc;
char *p = const_cast<char*>(pc); // ok: but writing through p is undefined
```

- 只有const_cast能改变表达式的常量属性，同样也不能用const_cast改变表达式的类型

```
const char *cp;
// error: static_cast can't cast away const
char *q = static_cast<char*>(cp);
static_cast<string>(cp); // ok: converts string literal to string
const_cast<string>(cp);  // error: const_cast only changes constness
```

reinterpret_cast

命名的强制类型转换

- reinterpret_cast通常为运算对象的位模式提供低层次上的重新解释
  - 使用reinterpret_cast是非常危险的

```
int *ip;
char *pc = reinterpret_cast<char*>(ip);
```

  - 必须牢记pc所指的真实对象是一个int而非字符，如果把pc当成普通的字符指针使用就可能在运行时发生错误

```
string str(pc);
```

旧式的强制类型转换

显式转换

- 在早期版本的C++中，显式类型转换包含两种形式：

  type (*expr*); // *function-style cast notation*
  (*type*) *expr*; // *C-language-style cast notation*

- 当在某处执行旧式的强制类型转换，如果换成const_cast或static_cast也合法，则其行为与对应的命名转换一致
- 如果替换后不合法，则旧式强制类型转换执行与reinterpret_cast类似的功能

```
int *ip;
char *pc = (char*) ip; // ip is a pointer to in
```