C++编程(4)

唐晓晟
北京邮电大学电信工程学院

第六章 表达式和语句

- 一个桌面计算器
- 运算符概览
- 语句概览
- 注释和缩进编排
- 忠告

6.1 一个桌面计算器

- 该计算器能够完成输入运算表达式的求解
- 由分析器、输入函数、符号表和一个驱动程序构成，可以看作是一个很小的编译器

6.2 运算符概览

- 在表格中，class_name表示类名，member表示一个成员的名字，object表示一个能产生出类对象的表达式，pointer是一个产生指针的表达式，expr是表达式，lvalue是一个表示非常量对象的表达式
- 表格中给出的只适合于内部类型的运算对象
- 每个间隔里的运算符具有相同优先级，上面间隔里面的运算符比下面间隔的运算符优先级更高

运算符概览(1)

作用域解析	class_name::member	全局	::name
作用域解析	namespace_name::member	全局	::qualified-name
成员选择	object.member	下标	pointer[expr]
成员选择	pointer->member	值构造	type(expr_list)
函数调用	expr(expr_list)	后增量	lvalue++
const转换	const_cast<type>(expr)	后减量	lvalue--
类型识别	typeid(type)	不检查的转换	reinterpret_cast<type>(expr)
运行时类型识别	type(expr)		
编译时检查的转换	static_cast<type>(expr)		
运行时检查的转换	dynamic_cast<type>(expr)		

运算符概览(2)

对象的大小	**sizeof expr**	前增量	**++lvalue**
类型的大小	**sizeof(type)**	前减量	**--lvalue**
地址	**&lvalue**	补	**~expr**
间接访问	***expr**	非(否定)	**!expr**
建立(分配)	**new type**	一元负号	**-expr**
建立(分配并初始化)	**new type(expr-list)**	一元正号	**+expr**
建立(放置)	**new (expr-list)type**		
建立(放置并初始化)	**new (expr-list)type(expr-list)**	强制(类型转换)	**(type)expr**
销毁(释放)	**delete pointer**	销毁数组	**delete []pointer**

运算符概览(3)

成员选择	object.*printer-to-member		
成员选择	pointer->*pointer-to-member		
乘	expr*expr	除	expr/expr
取模(余数)	expr%expr	加(求和)	expr+expr
减(求差)	expr-expr		
左移	expr << expr	右移	expr>>expr
小于	expr < expr	小于等于	expr <= expr
大于	expr > expr	大于等于	expr >= expr

运算符概览(4)

等于	**expr == expr**	不等于	**expr!=expr**
按位与	**expr&expr**		
按位异或	**expr^expr**		
按位或	**expr|expr**		
逻辑与	**expr&&expr**		
逻辑或	**expr||expr**		

运算符概览(5)

条件表达式	expr?expr:expr				
简单赋值	lvalue=expr			左移并赋值	lvalue<<=expr
乘并赋值	lvalue*=expr	加并赋值	lvalue+=expr	右移并赋值	lvalue>>=expr
除并赋值	lvalue/=expr	减并赋值	lvalue-=expr	异或并赋值	lvalue^=expr
与并赋值	lvalue&=expr	或并赋值	lvalue|=expr	取模并赋值	lvalue%=expr
抛出异常	throw expr				
逗号(序列)	expr,expr				

运算符示例

```
a+b*c //a+(b*c)
*p++  //先取得p所指向的数据的数值，然后p指针+1
```

一元运算符合赋值运算符是右结合的，其他运算符是左结合的

```
a=b=c // a=(b=c)
a+b+c // (a+b)+c
```

有不多的几条语法规则无法通过优先级和结合性来说明

```
a=b<c ? d=e : f=g
//a=((b<c)? (d=e) : (f=g))
```

6.2.1 结果

- 算术运算符的**结果类型**由一组称为“普通算术转换”的规则确定
  - 例如：二元运算符中有一个是浮点数，那么计算就将通过浮点算术进行；如果有一个long运算对象，那么计算用长整型算术，结果就是long，比int小的运算对象在运算符作用之前将被转换到int
- 关系运算符，==、<=等产生布尔值
- sizeof的结果是一个无符号整型size_t，指针减的结果是一个有符号整型ptrdiff_t，这些类型在<cstddef>中定义

结果示例

```
void f(int x, int y){
    int j = x = y;    // x = y的值是赋值后x的值
    int* p = &++x; // p指向x
    int* q = &(x++); // (t=x,x=x+1,t)
    //错误：x++不是一个左值(它不是存储在x里的值)
    int* pp = &(x>y ? x : y) //较大的那个int的地址
}
void f(){
    int i = 1;
    while( 0<i) i++;
    cout << “i has become negative!” << i << endl;
}
```

6.2.2 求值顺序

- 在一个表达式里，子表达式的求值顺序是没有定义的
- 运算符、逗号、&&、和||保证了位于它们左边的运算对象一定在右边运算对象之前求值(对于&&和||，注意短路算法)
- 括号可以用作强制性的结组

求值顺序示例

```
int x = f(2) + g(3) // 没定义f()和g()哪个先被调用

int i=1;
v[i] = i++; // 结果无定义

b = (a=2, a+1) // b = 3

f1(v[i], i++)     // 两个参数
f2( (v[i], i++) )// 一个参数 i++

a*b/c // (a*b)/c
// != a*(b/c)
```

6.2.3 运算符优先级

- 优先级层次和结合性影响到最普通的使用情况
- 对于程序员来说，如果对某些规则不清楚，就应该使用括号
- 编译器对于很多包含不规范甚至错误使用运算符的表达式可以提出警告

运算符优先级示例

```
if(i<=0 || max<i) // if( (i<=0) || (max <i))
// NOT if( i<= (0||max) < i)

if( i&mask ==0)
// if( i& (mask==0) ) 确实如此,20051012

if(0<=x<=99) //  if( (0<=x) <= 99)
// if( 0<=x && x<=99)

if(a=7) //条件中的常量赋值 应该用==
```

6.2.4 按位逻辑运算符

- 按位逻辑运算符&(交)、|(或)、^(异或)、~(补)、>>和<<可以应用于整型和枚举(bool, char, short, int, long, enum)
- 方便的位操作有可能非常重要，但是，为了可靠性、可维护性、可移植性等，应该将它保持在系统的底层中
- 位域也可以用于作为在一个机器字里移位和掩盖，以便抽取一段二进制位的方便方式

按位逻辑运算符示例

```
enum ios_base :: iostate {
    goodbit=0, eofbit=1, failbit=2, badbit=4
};

state = goodbit;
//...
if(state & (badbit|failbit)) // stream有问题

unsigned short middle(long a)
{
    return (a>>8) & 0xffff;
} //获取a中的第8bit到第23bit
```

6.2.5 增量和减量

- ++运算符用于直接表示相加，使人不必通过加和赋值的组合去间接地表示这个操作，例如：++lvalue意思是lvalue += 1，即lvalue = lvalue + 1。减量也类似地采用--运算符
- 运算符++和--都可以用作前缀或者后缀运算符，++x的值是(增量之后的)新值，例：y=++x等价于y=(x+=1)，x++的值则是x原有的值，例如：y=x++等价于y=(t=x,x+=1,t)，其中t是一个与x同类型的变量

增量和减量

- 运算符++和--也可以用到在对数组元素进行操作的指针上：p++，使p指向下一个元素

```
void cpy(char* p, const char* q){
    while(*p++ = *q++);
}

int length=strlen(q);
for(int i=0;i<=length;i++)
    p[i] = q[i]
// q被遍历了两遍，效率低
```

增量和减量示例

```
int i;
for(i=0;q[i]!=0;i++)
    p[i] = q[i];
p[i] = 0;
//用作下标的i可以去掉

while(*q!=0){
    *p = *q; p++; q++;
}
*p = 0;
//后增量运算符使我们可以先用
//变量的值，然后再增加它
```

```
while(*q!=0){
    *p++ = *q++;
}
*p = 0;
//由于*p++的值也就是*q

while ((*p++ = *q++)!=0)
{ }
//不需要空block
//也不需要和0进行比较

while(*p++=*q++);
```

6.2.6 自由存储

- 一般命名对象的生存时间由它的作用域决定
- 能够使用new操作符建立起生存时间不依赖于建立它作用域的对象，这类对象被称为是“在自由存储里的”或者是“堆对象”或者“在动态存储中分配”
- new创建的对象一直存在，直到使用delete运算符显式地销毁
- delete操作符只能用到new返回的指针或者0，对0操作不会造成任何影响

6.2.6.1 数组

- 可以用new建立对象的数组，比如：char* s = new char[32];
- 删除数组需要使用delete []，比如delete []s;
- 为了释放new分配的空间，delete和delete[]需要能够确定对象分配的空间大小，这意味着通过new分配的对象将会占用比静态对象稍微大一点的空间
- delete[]运算符只能应用于new返回的数组指针，应用到0不会产生任何影响

6.2.6.2 存储耗尽

- 自由存储运算符通过一些在头文件<new>里描述的函数实现：
  - void* operator new(size_t);
  - void operator delete(void*);
  - void* operator new[](size_t);
  - void operator delete[](void*);
- new的标准实现并不对返回的存储做初始化
- 当new无法找到分配的空间时，函数将抛出一个bad_alloc异常

存储耗尽示例

```
void f() {
    try { for(;;) new char[10000]; }
    catch(bad_alloc) { cerr << “Memory exhausted!”; }
}
```

我们也可以规定存储耗尽时new应该做什么

```
void out_of_store(){
    cerr << “operator new failed: out of store”;
    throw bad_alloc();
}
int main(){
    set_new_handler(out_of_store);
    for(;;) new char[10000];
    cout << “done!”;
}
```

6.2.7 显式类型转换

- 有时我们需要处理“原始的数据”，也就是那种保存或者将要保存某种对象的存储，而编译器并不知道对象的类型，例如：一个存储分配程序可能返回一个新分配存储块的void*指针，或者我们希望把某一个整数当成一个I/O设备的地址等

```
void* malloc(size_t);

void f() {
    int* p = static_cast<int*>(malloc(100));
    IO_device* dl = reinterpret_cast<IO_device*>(0xff00);
}
//显式类型转换
```

显式类型转换

- 显式类型转换常常被称做**强制**
- static_cast完成相关类型之间的转换，例如整型到枚举、浮点到整型等的转换
- reinterpret_cast处理互不相干类型之间的转换，例如从整型到指针等的转换
- 其他的类型转换形式还有dynamic_cast(运行中检查)和const_cast(清除const和volatile限定符)

显式类型转换

- C++从C中继承来T(e)记法，这种形式可以执行能用static_cast, reinterpret_cast, const_cast的组合表述的任何转换，它从表达式e出发去做一个T类型的值
- 这种C风格的强制远比上述的命名转换更危险，因为在大的程序里这种记法极难看清楚
- 总之，若感到要使用显式类型转换，请花一些时间考虑是否确实有必要

6.2.8 构造函数

- 从值e构造出一个类型T的值可以用函数记法T(e)表述，比如：int i = int(1.3)，这种结构有时被称作函数风格的强制
- 对于内部类型而言，T(e)等价于(T)e，不安全
- 构造函数记法T()用于描述类型T的默认值，例如：int j = int();
  complex z = complex();
- 对于内部类型，得到的是将0转换到该类型

6.3 语句概览

语句的语法

```
statement:
   declaration
   { statement-list_opt }
   try { statement-list_opt } handler-list
   expression_opt ;

   if( condition ) statement
   if( condition ) statement else statement
   switch ( condition ) statement

   while( condition ) statement
   do statement while ( expression ) ;
   for (for-init-statement condition_opt ; expression_opt ) statement
```

语句的语法

case constant-expression : statement
default : statement
break ;
continue ;

return expression$_{opt}$;

goto identifier ;
identifier : statement

statement-list:
statement statement-list$_{opt}$

语句的语法

condition:

expression

type-specifier declarator = expression

handler-list:

catch { exception-declaration } { statement-list$_{opt}$ }

handler-list handler-list$_{opt}$

6.3.1 声明作为语句

- 一个声明也是一个语句，除非一个变量被声明为static，否则它的初始式的执行就将在控制线程经过这个声明的时候进行
- 允许在所有写语句的地方写声明，是为了使程序员能够最大限度地减少由于未初始化的变量而导致的错误，并使得代码得到更好的局部化

6.3.2 选择语句

- if语句或者switch语句可以检测一个条件表达式的值，并以该值作为条件来决定需要执行的语句
- 比较运算符”== != < <= > >=”返回true或者false
- 逻辑运算符“&& ||”除了在必要时，不会去对第二个运算对象求值
- 有时，使用三目表达式或者switch语句能够得到更简洁的代码

选择语句示例

```
if(x) // ...
相当于if(x!=0) // ...

对于指针p
if(p) // ...
相当于 if(p!=0) // ...

if(p && p->count > 1)
// 短路算法

if(a<=b) max = b;
else max = a;

max = (a <= b) ? b : a;
```

```
switch (val) {
    case 1:
        f();
        break;
    case 2:
        g();
        break;
    default:
        h();
        break;
}
```

```
switch (val) {
    case 1:
        cout << “case 1”;
    case 2:
        cout << “case 2”;
    default:
        cout << “default:”;
}
```

6.3.2.1 在条件中的声明

- 为了避免意外地错误使用变量，在最小的作用域里引进变量是一个很好的想法，这样可以避免因为使用未初始化的变量而造成的麻烦
- 在条件中的声明只能声明和初始化单个的变量或者const

在条件中的声明示例

```
if( double d = prim(true)) {
    left /= d;
    break;
}

//上述原则的一个最优雅的应用
```

```
double d;
// ...
d2 = d;
// ...
if(d=prim(true)){
    left /= d;
    break;
}
// ...
d = 2.0;
//d的两个不相干的使用

//此为传统方式
```

6.3.3 迭代语句

- 循环语句可以用for,while或者do语句表述，这些语句都将反复执行一个称为受控语句或者循环体的语句，直到条件变为假，或者程序员要求以其他方式跳出该循环
- 根据作者经验，do语句是错误和混乱的一个根源

```
while ( condition ) statement
do statement while { expression };
for(for-init-statement; condition_opt; expression_opt)
    statement
```

6.3.3.1 for语句里的声明

- 可以在for语句的初始化部分中声明变量

```
void f(int v[], int max)
{
    for( int i = 0; i < max; i++)
        v[i] = i*i;
}
//i的值在for循环的整个过程中均有效
```

6.3.4 goto

- C++拥有臭名昭著的goto语句
- goto语句在高级程序设计中极少有用，但是在那些不是由人写的，而是由某个程序生成出来的C++代码就有可能很有用处
- 此外，在一些优化性能极端重要的程序中，goto也可能非常重要，例如：在某些实时应用的内存循环中
- 标号(label)的作用域是它所在的那个函数

6.4 注释和缩进编排

- 明智得使用注释、一致性地使用缩进编排形式，可以使阅读和理解一个程序的工作变得更轻松愉快
- 注释信息只对读者有用，编译器不会去理解注释的内容
- 糟糕的注释还不如没有

糟糕的注释示例

```
// 变量 “v” 必须初始化
// 变量 “v” 只能由函数 “f()” 使用
// 在调用这个文件中任何其他函数之前调用函数 “init()”
// 在你的程序最后调用函数 “cleanup()”
// 不要使用函数 “weird()”
// 函数 “f()” 有两个参数

a = b + c;  // a变为 b + c
count++;    // count 加1
```

作者的偏爱

- 为每个源文件写一个注释，一般性的陈述在它里面有哪些声明，对有关手册的引用，为维护而提供的一般性提示，如此等等
- 对每个类、模板和名字空间写一个注释
- 对每个非平凡的函数写一个注释，陈述其用途，所用的算法 (除非算法非常明显)，已经可能有的关于它对于环境所做的假设
- 对每个全局的和名字空间的变量和常量写一个注释
- 在非明显或不可移植的代码处的少量注释
- 极少其他的东西

6.5 忠告

- [1]应尽可能使用标准库，而不是其他的库和“手工打造的代码”；
- [2]避免过于复杂的表达式；
- [3]如果对运算符的优先级有疑问，加括号；
- [4]避免显式类型转换 (强制)；
- [5]若必须做显式类型转换，提倡使用特殊强制运算符，而不是C风格的强制；
- [6]只对定义良好的构造使用T(e)记法；
- [7]避免带有无定义求值顺序的表达式；
- [8]避免goto语句；

忠告

- [9]避免do语句；
- [10]声明变量时，最好同时进行初始化；
- [11]使注释简洁、清晰、有意义；
- [12]保持一致的缩进编排风格；
- [13]倾向于去定义一个成员函数operator new()去取代全局的operator new()；
- [14]在读输入的时候，总应考虑病态形式的输入；