第八章 表达式和基本语句

读者可能怀疑：连 if、for、while、goto、switch 这样简单的东西也要探讨编程风格，是不是小题大做？

我真的发觉很多程序员用隐含错误的方式写表达式和基本语句，我自己也犯过类似的错误。

表达式和语句都属于 C++/C 的短语结构语法。它们看似简单，但使用时隐患比较多。本章归纳了正确使用表达式和语句的一些规则与建议。

8.1 运算符的优先级

C++/C 语言的运算符有数十个，运算符的优先级与结合律如表 8-1 所示。注意一元运算符 + - * 的优先级高于对应的二元运算符。

优先级	运算符	结合律
从高到低排列	( ) [ ] -> .	从左至右
	! ~ ++ -- （类型） sizeof + - * &	**从右至左**
	* / %	从左至右
	+ -	从左至右
	<< >>	从左至右
	< <= > >=	从左至右
	== !=	从左至右
	&	从左至右
	^	从左至右
	|	从左至右
	&&	从左至右
	||	**从右至左**
	?:	**从右至左**
	= += -= *= /= %= &= ^= |= <<= >>=	从左至右

表 8-1 运算符的优先级与结合律

- **【规则 8-1-1】**如果代码行中的运算符比较多，用括号确定表达式的操作顺序，避免使用默认的优先级。

  由于将表 8-1 熟记是比较困难的，为了防止产生歧义并提高可读性，应当用括号确

定表达式的操作顺序。例如：

```
word = (high << 8) | low
if ((a | b) && (a & c))
```

8.2 复合表达式

如 a = b = c = 0 这样的表达式称为复合表达式。允许复合表达式存在的理由是：（1）书写简洁；（2）可以提高编译效率。但要防止滥用复合表达式。

- **【规则 8-2-1】**不要编写太复杂的复合表达式。

  例如：

```
i = a >= b && c < d && c + f <= g + h ;   // 复合表达式过于复杂
```

- **【规则 8-2-2】**不要有多用途的复合表达式。

  例如：

```
d = (a = b + c) + r ;
```

  该表达式既求 a 值又求 d 值。应该拆分为两个独立的语句：

```
a = b + c;
d = a + r;
```

- **【规则 8-2-3】**不要把程序中的复合表达式与“真正的数学表达式”混淆。

  例如：

```
if (a < b < c)          // a < b < c 是数学表达式而不是程序表达式
```

  并不表示

```
if ((a<b) && (b<c))
```

  而是成了令人费解的

```
if ( (a<b)<c )
```

8.3 if 语句

if 语句是 C++/C 语言中最简单、最常用的语句，然而很多程序员用隐含错误的方式写 if 语句。本节以“与零值比较”为例，展开讨论。

8.3.1 布尔变量与零值比较

- **【规则 8-3-1】**不可将布尔变量直接与 TRUE、FALSE 或者 1、0 进行比较。

  根据布尔类型的语义，零值为“假”（记为 FALSE），任何非零值都是“真”（记为

TRUE)。TRUE 的值究竟是什么并没有统一的标准。例如 Visual C++ 将 TRUE 定义为 1，而 Visual Basic 则将 TRUE 定义为-1。

假设布尔变量名字为 flag，它与零值比较的标准 if 语句如下：

```
if (flag)  // 表示 flag 为真
if (!flag) // 表示 flag 为假
```

其它的用法都属于不良风格，例如：

```
if (flag == TRUE)
if (flag == 1 )
if (flag == FALSE)
if (flag == 0)
```

8.3.2 整型变量与零值比较

- **【规则 8-3-2】**应当将整型变量用“==”或“！=”直接与 0 比较。

假设整型变量的名字为 value，它与零值比较的标准 if 语句如下：

```
if (value == 0)
if (value != 0)
```

不可模仿布尔变量的风格而写成

```
if (value)     // 会让人误解 value 是布尔变量
if (!value)
```

8.3.3 浮点变量与零值比较

- **【规则 8-3-3】**不可将浮点变量用“==”或“！=”与任何数字比较。

千万要留意，无论是 float 还是 double 类型的变量，都有精度限制。所以一定要避免将浮点变量用“==”或“！=”与数字比较，应该设法转化成“>=”或“<=”形式。

假设浮点变量的名字为 x，应当将

```
if (x == 0.0)  // 隐含错误的比较
```

转化为

```
if ((x>=-EPSINON) && (x<=EPSINON))
```

其中 EPSINON 是允许的误差（即精度）。

8.3.4 指针变量与零值比较

- **【规则 8-3-4】**应当将指针变量用“==”或“！=”与 NULL 比较。

指针变量的零值是“空”（记为 NULL）。尽管 NULL 的值与 0 相同，但是两者意义不同。假设指针变量的名字为 p，它与零值比较的标准 if 语句如下：

```
if (p == NULL) // p 与 NULL 显式比较，强调 p 是指针变量
```

```
    if (p != NULL)
```

不要写成

```
    if (p == 0)     // 容易让人误解 p 是整型变量
    if (p != 0)
```

或者

```
    if (p)          // 容易让人误解 p 是布尔变量
    if (!p)
```

8.3.5 对 if 语句的补充说明

有时候我们可能会看到 if (NULL == p) 这样古怪的格式。不是程序写错了，而是程序员为了防止将 if (p == NULL) 误写成 if (p = NULL)，有意把 p 和 NULL 颠倒。编译器认为 if (p = NULL) 是合法的，但是会指出 if (NULL = p)是错误的，因为 NULL 不能被赋值。

程序中有时会遇到 if/else/return 的组合，应该将如下不良风格的程序

```
    if (condition)
        return x;
    return y;
```

改写为

```
    if (condition)
    {
        return x;
    }
    else
    {
        return y;
    }
```

或者改写成更加简练的

```
    return (condition ? x : y);
```

8.4 循环语句的效率

C++/C 循环语句中，for 语句使用频率最高，while 语句其次，do 语句很少用。本节重点论述循环体的效率。提高循环体效率的基本办法是降低循环体的复杂性。

- **【建议 8-4-1】**在多重循环中，如果有可能，应当将最长的循环放在最内层，最短的循环放在最外层，以减少 CPU 跨切循环层的次数。例如示例 8-4(b)的效率比示例 8-4(a)的高。

```
for (row=0; row<100; row++)
{
    for ( col=0; col<5; col++ )
    {
        sum = sum + a[row][col];
    }
}
```

示例 8-4(a) 低效率：长循环在最外层

```
for (col=0; col<5; col++ )
{
    for (row=0; row<100; row++)
    {
        sum = sum + a[row][col];
    }
}
```

示例 8-4(b) 高效率：长循环在最内层

- **【建议 8-4-2】**如果循环体内存在逻辑判断，并且循环次数很大，宜将逻辑判断移到循环体的外面。

示例 8-4(c)的程序比示例 8-4(d)多执行了 N-1 次逻辑判断。并且由于前者老要进行逻辑判断，打断了循环“流水线”作业，使得编译器不能对循环进行优化处理，降低了效率。如果 N 非常大，最好采用示例 8-4(d)的写法，可以提高效率。如果 N 非常小，两者效率差别并不明显，采用示例 8-4(c)的写法比较好，因为程序更加简洁。

```
for (i=0; i<N; i++)
{
  if (condition)
      DoSomething();
  else
      DoOtherthing();
}
```

表 8-4(c) 效率低但程序简洁

```
if (condition)
{
    for (i=0; i<N; i++)
      DoSomething();
}
else
{
    for (i=0; i<N; i++)
      DoOtherthing();
}
```

表 8-4(d) 效率高但程序不简洁

8.5 for 语句的循环控制变量

- **【规则 8-5-1】**不可在 for 循环体内修改循环变量，防止 for 循环失去控制。

- **【建议 8-5-1】**建议 for 语句的循环控制变量的取值采用“半开半闭区间”写法。

示例 8-5(a)中的 x 值属于半开半闭区间“0 =< x < N”，起点到终点的间隔为 N，循环次数为 N。

示例 8-5(b)中的 x 值属于闭区间“0 =< x <= N-1”，起点到终点的间隔为 N-1，循环次数为 N。

相比之下，示例 8-5(a)的写法更加直观，尽管两者的功能是相同的。

```
for (int x=0; x<N; x++)
{
  …
}
```

示例 8-5(a) 循环变量属于半开半闭区间

```
for (int x=0; x<=N-1; x++)
{
  …
}
```

示例 8-5(b) 循环变量属于闭区间

8.6 switch 语句

有了 if 语句为什么还要 switch 语句?

switch 是多分支选择语句,而 if 语句只有两个分支可供选择。虽然可以用嵌套的 if 语句来实现多分支选择，但那样的程序冗长难读。这是 switch 语句存在的理由。

switch 语句的基本格式是：

```
switch (variable)
{
    case value1 :  …
                   break;
    case value2 :  …
                   break;
    …
        default :  …
                   break;
}
```

- **【规则 8-6-1】**每个 case 语句的结尾不要忘了加 break，否则将导致多个分支重叠（除非有意使多个分支重叠)。
- **【规则 8-6-2】**不要忘记最后那个 default 分支。即使程序真的不需要 default 处理，也应该保留语句 default : break; 这样做并非多此一举，而是为了防止别人误以为你忘了 default 处理。

8.7 goto 语句

自从提倡结构化设计以来，goto 就成了有争议的语句。首先，由于 goto 语句可以灵活跳转，如果不加限制，它的确会破坏结构化设计风格。其次，goto 语句经常带来错误或隐患。它可能跳过了某些对象的构造、变量的初始化、重要的计算等语句，例如：

```
goto state;
String s1, s2; // 被 goto 跳过
int sum = 0;   // 被 goto 跳过
…
state:
…
```

如果编译器不能发觉此类错误，每用一次 goto 语句都可能留下隐患。

很多人建议废除 C++/C 的 goto 语句，以绝后患。但实事求是地说，错误是程序员自己造成的，不是 goto 的过错。goto 语句至少有一处可显神通，它能从多重循环体中咻地一下子跳到外面，用不着写很多次的 break 语句; 例如

```
{ …
    { …
        { …
            goto error;
        }
    }
}
error:
…
```

就象楼房着火了，来不及从楼梯一级一级往下走，可从窗口跳出火坑。所以我们主张少用、慎用 goto 语句，而不是禁用。