第十五章 其它编程经验

15.1 使用 const 提高函数的健壮性

看到 const 关键字，C++程序员首先想到的可能是 const 常量。这可不是良好的条件反射。如果只知道用 const 定义常量，那么相当于把火药仅用于制作鞭炮。const 更大的魅力是它可以修饰函数的参数、返回值，甚至函数的定义体。

const 是 constant 的缩写，“恒定不变”的意思。被 const 修饰的东西都受到强制保护，可以预防意外的变动，能提高程序的健壮性。所以很多 C++程序设计书籍建议：“Use const whenever you need”。

15.1.1 用 const 修饰函数的参数

如果参数作输出用，不论它是什么数据类型，也不论它采用“指针传递”还是“引用传递”，都不能加 const 修饰，否则该参数将失去输出功能。

const 只能修饰输入参数：

◆ 如果输入参数采用“指针传递”，那么加 const 修饰可以防止意外地改动该指针，起到保护作用。

例如 StringCopy 函数：

```
void StringCopy(char *strDestination, const char *strSource);
```

其中 strSource 是输入参数，strDestination 是输出参数。给 strSource 加上 const 修饰后，如果函数体内的语句试图改动 strSource 的内容，编译器将指出错误。

◆ 如果输入参数采用“值传递”，由于函数将自动产生临时变量用于复制该参数，该输入参数本来就无需保护，所以不要加 const 修饰。

例如不要将函数 void Func1(int x) 写成 void Func1(const int x)。同理不要将函数 void Func2(A a) 写成 void Func2(const A a)。其中 A 为用户自定义的数据类型。

◆ 对于非内部数据类型的参数而言，象 void Func(A a) 这样声明的函数注定效率比较底。因为函数体内将产生 A 类型的临时对象用于复制参数 a，而临时对象的构造、复制、析构过程都将消耗时间。

为了提高效率，可以将函数声明改为 void Func(A &a)，因为“引用传递”仅借用一下参数的别名而已，不需要产生临时对象。但是函数 void Func(A &a) 存在一个缺点：“引用传递”有可能改变参数 a，这是我们不期望的。解决这个问题很容易，加 const 修饰即可，因此函数最终成为 void Func(const A &a)。

以此类推，是否应将 void Func(int x) 改写为 void Func(const int &x)，以便提

高效率？完全没有必要，因为内部数据类型的参数不存在构造、析构的过程，而复制也非常快，“值传递”和“引用传递”的效率几乎相当。

问题是如此的缠绵，我只好将“const &”修饰输入参数的用法总结一下，如表 15-1-1 所示。

对于非内部数据类型的输入参数，应该将“值传递”的方式改为“const 引用传递”，目的是提高效率。例如将 void Func(A a) 改为 void Func(const A &a)。
对于内部数据类型的输入参数，不要将“值传递”的方式改为“const 引用传递”。否则既达不到提高效率的目的，又降低了函数的可理解性。例如 void Func(int x) 不应该改为 void Func(const int &x)。

表 15-1-1 “const &”修饰输入参数的规则

15.1.2 用 const 修饰函数的返回值

◆ 如果给以“指针传递”方式的函数返回值加 const 修饰，那么函数返回值（即指针）的内容不能被修改，该返回值只能被赋给加 const 修饰的同类型指针。

例如函数

```
const char * GetString(void);
```

如下语句将出现编译错误：

```
char *str = GetString();
```

正确的用法是

```
const char *str = GetString();
```

◆ 如果函数返回值采用“值传递方式”，由于函数会把返回值复制到外部临时的存储单元中，加 const 修饰没有任何价值。

例如不要把函数 int GetInt(void) 写成 const int GetInt(void)。

同理不要把函数 A GetA(void) 写成 const A GetA(void)，其中 A 为用户自定义的数据类型。

如果返回值不是内部数据类型，将函数 A GetA(void) 改写为 const A & GetA(void)的确能提高效率。但此时千万千万要小心，一定要搞清楚函数究竟是想返回一个对象的“拷贝”还是仅返回“别名”就可以了，否则程序会出错。见 15.2 节“返回值的规则”。

◆ 函数返回值采用“引用传递”的场合并不多，这种方式一般只出现在类的赋值函数中，目的是为了实现链式表达。

例如

```
class A
```

```
{…
    A & operate = (const A &other);   // 赋值函数
};
A a, b, c;      // a, b, c 为 A 的对象
…
a = b = c;          // 正常的链式赋值
(a = b) = c;        // 不正常的链式赋值，但合法
```

如果将赋值函数的返回值加 const 修饰，那么该返回值的内容不允许被改动。上例中，语句 a = b = c 仍然正确，但是语句 (a = b) = c 则是非法的。

15.1.3 const 成员函数

任何不会修改数据成员的函数都应该声明为 const 类型。如果在编写 const 成员函数时，不慎修改了数据成员，或者调用了其它非 const 成员函数，编译器将指出错误，这无疑会提高程序的健壮性。

以下程序中，类 stack 的成员函数 GetCount 仅用于计数，从逻辑上讲 GetCount 应当为 const 函数。编译器将指出 GetCount 函数中的错误。

```
class Stack
{
  public:
    void    Push(int elem);
    int     Pop(void);
    int     GetCount(void)  const; // const 成员函数
  private:
    int     m_num;
    int     m_data[100];
};

int Stack::GetCount(void)  const
{
    ++ m_num;  // 编译错误，企图修改数据成员 m_num
    Pop();     // 编译错误，企图调用非 const 函数
    return m_num;
}
```

const 成员函数的声明看起来怪怪的：const 关键字只能放在函数声明的尾部，大概是因为其它地方都已经被占用了。

15.2 提高程序的效率

程序的时间效率是指运行速度，空间效率是指程序占用内存或者外存的状况。

全局效率是指站在整个系统的角度上考虑的效率，局部效率是指站在模块或函数角度上考虑的效率。

- **【规则 15-2-1】**不要一味地追求程序的效率，应当在满足正确性、可靠性、健壮性、可读性等质量因素的前提下，设法提高程序的效率。

- **【规则 15-2-2】**以提高程序的全局效率为主，提高局部效率为辅。

- **【规则 15-2-3】**在优化程序的效率时，应当先找出限制效率的“瓶颈”，不要在无关紧要之处优化。

- **【规则 15-2-4】**先优化数据结构和算法，再优化执行代码。

- **【规则 15-2-5】**有时候时间效率和空间效率可能对立，此时应当分析那个更重要，作出适当的折衷。例如多花费一些内存来提高性能。

- **【规则 15-2-6】**不要追求紧凑的代码，因为紧凑的代码并不能产生高效的机器码。

15.3 一些有益的建议

✧ **【建议 15-3-1】**当心那些视觉上不易分辨的操作符发生书写错误。
我们经常会把“＝＝”误写成“＝”，象“||”、“&&”、“<=”、“>=”这类符号也很容易发生“丢 1”失误。然而编译器却不一定能自动指出这类错误。

✧ **【建议 15-3-2】**变量（指针、数组）被创建之后应当及时把它们初始化，以防止把未被初始化的变量当成右值使用。

✧ **【建议 15-3-3】**当心变量的初值、缺省值错误，或者精度不够。

✧ **【建议 15-3-4】**当心数据类型转换发生错误。尽量使用显式的数据类型转换（让人们知道发生了什么事），避免让编译器轻悄悄地进行隐式的数据类型转换。

✧ **【建议 15-3-5】**当心变量发生上溢或下溢，数组的下标越界。

✧ **【建议 15-3-6】**当心忘记编写错误处理程序，当心错误处理程序本身有误。

✧ **【建议 15-3-7】**当心文件 I/O 有错误。

✧ **【建议 15-3-8】**避免编写技巧性很高代码。

✧ **【建议 15-3-9】**不要设计面面俱到、非常灵活的数据结构。

✧ **【建议 15-3-10】**如果原有的代码质量比较好，尽量复用它。但是不要修补很差劲的代码，应当重新编写。

✧ **【建议 15-3-11】**尽量使用标准库函数，不要“发明”已经存在的库函数。

✧ **【建议 15-3-12】**尽量不要使用与具体硬件或软件环境关系密切的变量。

✧ **【建议 15-3-13】**把编译器的选择项设置为最严格状态。

✧ **【建议 15-3-14】**如果可能的话，使用 PC-Lint、LogiScope 等工具进行代码审查。