第十章 函数设计

函数是 C++/C 程序的基本功能单元，其重要性不言而喻。函数设计的细微缺点很容易导致该函数被错用，所以光使函数的功能正确是不够的。本章重点论述函数的接口设计和内部实现的一些规则。

函数接口的两个要素是参数和返回值。C 语言中，函数的参数和返回值的传递方式有两种：值传递（pass by value）和指针传递（pass by pointer）。C++ 语言中多了引用传递（pass by reference）。由于引用传递的性质象指针传递，而使用方式却象值传递，初学者常常迷惑不解，容易引起混乱，请先阅读 10.6 节“引用与指针的比较”。

10.1 参数的规则

- **【规则 10-1-1】**参数的书写要完整，不要贪图省事只写参数的类型而省略参数名字。

  如果函数没有参数，则用 void 填充。

  例如：

```
void SetValue(int width, int height); // 良好的风格
void SetValue(int, int);              // 不良的风格
float GetValue(void);      // 良好的风格
float GetValue();          // 不良的风格
```

- **【规则 10-1-2】**参数命名要恰当，顺序要合理。

  编写字符串拷贝函数 StringCopy。它有两个参数，如果把参数名字起为 str1 和 str2，例如：

```
void StringCopy(char *str1, char *str2);
```

  那么我们很难搞清楚究竟是把 str1 拷贝到 str2 中，还是刚好倒过来。

  可以把参数名字起得更有意义，如叫 strSource 和 strDestination。这样从名字上就可以看出应该把 strSource 拷贝到 strDestination。

  还有一个问题，这两个参数那一个该在前那一个该在后？参数的顺序要遵循程序员的习惯。一般地，应将目的参数放在前面，源参数放在后面。

  如果将函数声明为：

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

  别人在使用时可能会不假思索地写成如下形式：

```
char str[20];
StringCopy(str, “Hello World”); // 参数顺序颠倒
```

- **【规则 10-1-3】**如果参数是指针，且仅作输入用，则应在类型前加 const，以防止该指针在函数体内被意外修改。

例如：

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

- 【规则 10-1-4】如果输入参数以值传递的方式传递对象，则宜改用“const &”方式来传递，这样可以省去临时对象的构造和析构过程，从而提高效率。

✧ 【建议 10-1-1】避免函数有太多的参数，参数个数尽量控制在 5 个以内。如果参数太多，在使用时容易将参数类型或顺序搞错。

✧ 【建议 10-1-2】尽量不要使用类型和数目不确定的参数。

C 标准库函数 printf 是采用不确定参数的典型代表，其原型为：

```
int printf(const chat *format[, argument]…);
```

这种风格的函数在编译时丧失了严格的类型安全检查。

10.2 返回值的规则

- 【规则 10-2-1】不要省略返回值的类型。

C 语言中，凡不加类型说明的函数，一律自动按整型处理。这样做不会有什么好处，却容易被误解为 void 类型。

C++语言有很严格的类型安全检查，不允许上述情况发生。由于 C++程序可以调用 C 函数，为了避免混乱，规定任何 C++/ C 函数都必须有类型。如果函数没有返回值，那么应声明为 void 类型。

- 【规则 10-2-2】函数名字与返回值类型在语义上不可冲突。

违反这条规则的典型代表是 C 标准库函数 getchar。

例如：

```
char c;
c = getchar();
if (c == EOF)
…
```

按照 getchar 名字的意思，将变量 c 声明为 char 类型是很自然的事情。但不幸的是 getchar 的确不是 char 类型，而是 int 类型，其原型如下：

```
int getchar(void);
```

由于 c 是 char 类型，取值范围是[-128，127]，如果宏 EOF 的值在 char 的取值范围之外，那么 if 语句将总是失败，这种“危险”人们一般哪里料得到！导致本例错误的责任并不在用户，是函数 getchar 误导了使用者。

- 【规则 10-2-3】不要将正常值和错误标志混在一起返回。正常值用输出参数获得，而错误标志用 return 语句返回。

回顾上例，C 标准库函数的设计者为什么要将 getchar 声明为令人迷糊的 int 类型呢？他会那么傻吗？

在正常情况下，getchar 的确返回单个字符。但如果 getchar 碰到文件结束标志或发生读错误，它必须返回一个标志 EOF。为了区别于正常的字符，只好将 EOF 定义为负数（通常为负 1）。因此函数 getchar 就成了 int 类型。

我们在实际工作中，经常会碰到上述令人为难的问题。为了避免出现误解，我们应该将正常值和错误标志分开。即：正常值用输出参数获得，而错误标志用 return 语句返回。

函数 getchar 可以改写成 BOOL GetChar(char *c);

虽然 gechar 比 GetChar 灵活，例如 putchar(getchar()); 但是如果 getchar 用错了，它的灵活性又有什么用呢？

✧ **【建议 10-2-1】**有时候函数原本不需要返回值，但为了增加灵活性如支持链式表达，可以附加返回值。

例如字符串拷贝函数 strcpy 的原型：

```
char *strcpy(char *strDest，const char *strSrc);
```

strcpy 函数将 strSrc 拷贝至输出参数 strDest 中，同时函数的返回值又是 strDest。这样做并非多此一举，可以获得如下灵活性：

```
char str[20];
int  length = strlen( strcpy(str, “Hello World”) );
```

✧ **【建议 10-2-2】**如果函数的返回值是一个对象，有些场合用“引用传递”替换“值传递”可以提高效率。而有些场合只能用“值传递”而不能用“引用传递”，否则会出错。

例如：

```
class String
{…
    // 赋值函数
    String & operate=(const String &other);
    // 相加函数，如果没有 friend 修饰则只许有一个右侧参数
    friend  String   operate+( const String &s1, const String &s2);
private:
    char *m_data;
}
```

String 的赋值函数 operate = 的实现如下：

```
String & String::operate=(const String &other)
{
    if (this == &other)
        return *this;
```

```
    delete m_data;
    m_data = new char[strlen(other.data)+1];
    strcpy(m_data, other.data);
    return *this;  // 返回的是 *this 的引用，无需拷贝过程
}
```

对于赋值函数，应当用“引用传递”的方式返回 String 对象。如果用“值传递”的方式，虽然功能仍然正确，但由于 return 语句要把 *this 拷贝到保存返回值的外部存储单元之中，增加了不必要的开销，降低了赋值函数的效率。例如：

```
String a,b,c;
…
a = b;      // 如果用“值传递”，将产生一次 *this 拷贝
a = b = c;  // 如果用“值传递”，将产生两次 *this 拷贝
```

String 的相加函数 operate + 的实现如下：

```
String  operate+(const String &s1, const String &s2)
{
    String temp;
    delete temp.data;  // temp.data 是仅含‘\0’的字符串
    temp.data = new char[strlen(s1.data) + strlen(s2.data) +1];
    strcpy(temp.data, s1.data);
    strcat(temp.data, s2.data);
    return temp;
}
```

对于相加函数，应当用“值传递”的方式返回 String 对象。如果改用“引用传递”，那么函数返回值是一个指向局部对象 temp 的“引用”。由于 temp 在函数结束时被自动销毁，将导致返回的“引用”无效。例如：

```
c = a + b;
```

此时 a + b 并不返回期望值，c 什么也得不到，流下了隐患。

10.3 函数内部实现的规则

不同功能的函数其内部实现各不相同，看起来似乎无法就“内部实现”达成一致的观点。但根据经验，我们可以在函数体的“入口处”和“出口处”从严把关，从而提高函数的质量。

- **【规则 10-3-1】**在函数体的“入口处”，对参数的有效性进行检查。

  很多程序错误是由非法参数引起的，我们应该充分理解并正确使用“断言”（assert）

来防止此类错误。详见 10.5 节“使用断言”。

● 【规则 10-3-2】在函数体的“出口处”，对 return 语句的正确性和效率进行检查。

如果函数有返回值，那么函数的“出口处”是 return 语句。我们不要轻视 return 语句。如果 return 语句写得不好，函数要么出错，要么效率低下。

注意事项如下：

（1）return 语句不可返回指向“栈内存”的“指针”或者“引用”，因为该内存在函数体结束时被自动销毁。例如

```
char * Func(void)
{
    char str[] = “hello world”;// str 的内存位于栈上
    …
    return str;     // 将导致错误
}
```

（2）要搞清楚返回的究竟是“值”、“指针”还是“引用”。

（3）如果函数返回值是一个对象，要考虑 return 语句的效率。例如

```
return String(s1 + s2);
```

这是临时对象的语法，表示“创建一个临时对象并返回它”。不要以为它与“先创建一个局部对象 temp 并返回它的结果”是等价的，如

```
String temp(s1 + s2);
return temp;
```

实质不然，上述代码将发生三件事。首先，temp 对象被创建，同时完成初始化；然后拷贝构造函数把 temp 拷贝到保存返回值的外部存储单元中；最后，temp 在函数结束时被销毁（调用析构函数）。然而“创建一个临时对象并返回它”的过程是不同的，编译器直接把临时对象创建并初始化在外部存储单元中，省去了拷贝和析构的化费，提高了效率。

类似地，我们不要将

```
return int(x + y); // 创建一个临时变量并返回它
```

写成

```
int temp = x + y;
return temp;
```

由于内部数据类型如 int,float,double 的变量不存在构造函数与析构函数，虽然该“临时变量的语法”不会提高多少效率，但是程序更加简洁易读。

10.4 其它建议

✧ 【建议 10-4-1】函数的功能要单一，不要设计多用途的函数。

✧ 【建议 10-4-2】函数体的规模要小，尽量控制在 50 行代码之内。

✧ **【建议 10-4-3】**尽量避免函数带有“记忆”功能。相同的输入应当产生相同的输出。

带有“记忆”功能的函数，其行为可能是不可预测的，因为它的行为可能取决于某种“记忆状态”。这样的函数既不易理解又不利于测试和维护。在 C/C++语言中，函数的 static 局部变量是函数的“记忆”存储器。建议尽量少用 static 局部变量，除非必需。

✧ **【建议 10-4-4】**不仅要检查输入参数的有效性，还要检查通过其它途径进入函数体内的变量的有效性，例如全局变量、文件句柄等。

✧ **【建议 10-4-5】**用于出错处理的返回值一定要清楚，让使用者不容易忽视或误解错误情况。

10.5 使用断言

程序一般分为 Debug 版本和 Release 版本，Debug 版本用于内部调试，Release 版本发行给用户使用。

断言 assert 是仅在 Debug 版本起作用的宏，它用于检查“不应该”发生的情况。示例 10-5 是一个内存复制函数。在运行过程中，如果 assert 的参数为假，那么程序就会中止（一般地还会出现提示对话，说明在什么地方引发了 assert）。

```
void  *memcpy(void *pvTo, const void *pvFrom, size_t size)
{
    assert((pvTo != NULL) && (pvFrom != NULL));    // 使用断言
    byte *pbTo = (byte *) pvTo;          // 防止改变 pvTo 的地址
    byte *pbFrom = (byte *) pvFrom;      // 防止改变 pvFrom 的地址
    while(size -- > 0 )
        *pbTo ++ = *pbFrom ++ ;
    return pvTo;
}
```

示例 10-5 复制不重叠的内存块

assert 不是一个仓促拼凑起来的宏。为了不在程序的 Debug 版本和 Release 版本引起差别，assert 不应该产生任何副作用。所以 assert 不是函数，而是宏。程序员可以把 assert 看成一个在任何系统状态下都可以安全使用的无害测试手段。**如果程序在 assert 处终止了，并不是说含有该 assert 的函数有错误，而是调用者出了差错，assert 可以帮助我们找到发生错误的原因。**

很少有比跟踪到程序的断言，却不知道该断言的作用更让人沮丧的事了。你化了很多时间，不是为了排除错误，而只是为了弄清楚这个错误到底是什么。有的时候，程序

员偶尔还会设计出有错误的断言。所以如果搞不清楚断言检查的是什么，就很难判断错误是出现在程序中，还是出现在断言中。幸运的是这个问题很好解决，只要加上清晰的注释即可。这本是显而易见的事情，可是很少有程序员这样做。这好比一个人在森林里，看到树上钉着一块“危险”的大牌子。但危险到底是什么？树要倒？有废井？有野兽？除非告诉人们“危险”是什么，否则这个警告牌难以起到积极有效的作用。难以理解的断言常常被程序员忽略，甚至被删除。[Maguire93, p8-p30]

- **【规则 10-5-1】**使用断言捕捉不应该发生的非法情况。不要混淆非法情况与错误情况之间的区别，后者是必然存在的并且是一定要作出处理的。

- **【规则 10-5-2】在**函数的入口处，使用断言检查参数的有效性（合法性）。

- **【建议 10-5-1】**在编写函数时，要进行反复的考查，并且自问：“我打算做哪些假定？”一旦确定了的假定，就要使用断言对假定进行检查。

- **【建议 10-5-2】**一般教科书都鼓励程序员们进行防错设计，但要记住这种编程风格可能会隐瞒错误。当进行防错设计时，如果“不可能发生”的事情的确发生了，则要使用断言进行报警。

10.6 引用与指针的比较

引用是 C++中的概念，初学者容易把引用和指针混淆一起。一下程序中，n 是 m 的一个引用（reference），m 是被引用物（referent）。

```
int m;
int &n = m;
```

n 相当于 m 的别名（绰号），对 n 的任何操作就是对 m 的操作。例如有人名叫王小毛，他的绰号是“三毛”。说“三毛”怎么怎么的，其实就是对王小毛说三道四。所以 n 既不是 m 的拷贝，也不是指向 m 的指针，其实 n 就是 m 它自己。

引用的一些规则如下：

（1）引用被创建的同时必须被初始化（指针则可以在任何时候被初始化）。

（2）不能有 NULL 引用，引用必须与合法的存储单元关联（指针则可以是 NULL）。

（3）一旦引用被初始化，就不能改变引用的关系（指针则可以随时改变所指的对象）。

以下示例程序中，k 被初始化为 i 的引用。语句 k = j 并不能将 k 修改成为 j 的引用，只是把 k 的值改变成为 6。由于 k 是 i 的引用，所以 i 的值也变成了 6。

```
int i = 5;
int j = 6;
int &k = i;
k = j;  // k 和 i 的值都变成了 6;
```

上面的程序看起来象在玩文字游戏，没有体现出引用的价值。引用的主要功能是传

递函数的参数和返回值。C++语言中，函数的参数和返回值的传递方式有三种：值传递、指针传递和引用传递。

以下是“值传递”的示例程序。由于 Func1 函数体内的 x 是外部变量 n 的一份拷贝，改变 x 的值不会影响 n，所以 n 的值仍然是 0。

```
void Func1(int x)
{
    x = x + 10;
}
…
int n = 0;
Func1(n);
cout << “n = ” << n << endl; // n = 0
```

以下是“指针传递”的示例程序。由于 Func2 函数体内的 x 是指向外部变量 n 的指针，改变该指针的内容将导致 n 的值改变，所以 n 的值成为 10。

```
void Func2(int *x)
{
    (* x) = (* x) + 10;
}
…
int n = 0;
Func2(&n);
cout << “n = ” << n << endl;     // n = 10
```

以下是“引用传递”的示例程序。由于 Func3 函数体内的 x 是外部变量 n 的引用，x 和 n 是同一个东西，改变 x 等于改变 n，所以 n 的值成为 10。

```
void Func3(int &x)
{
    x = x + 10;
}
…
int n = 0;
Func3(n);
cout << “n = ” << n << endl;     // n = 10
```

对比上述三个示例程序，会发现“引用传递”的性质象“指针传递"，而书写方式象“值传递"。实际上“引用”可以做的任何事情“指针”也都能够做，为什么还要“引用"这东西？

答案是“用适当的工具做恰如其分的工作"。

指针能够毫无约束地操作内存中的如何东西，尽管指针功能强大，但是非常危险。

就象一把刀，它可以用来砍树、裁纸、修指甲、理发等等，谁敢这样用？

如果的确只需要借用一下某个对象的“别名”，那么就用“引用”，而不要用“指针”，以免发生意外。比如说，某人需要一份证明，本来在文件上盖上公章的印子就行了，如果把取公章的钥匙交给他，那么他就获得了不该有的权利。