

第 1 章 假想的编译程序

读者可以考虑一下倘若编译程序能够正确地指出代码中的所有问题,那相应程序的错误情况会怎样?这不但指语法错误,还包括程序中的任何问题,不管它有多么隐蔽。例如,假定程序中有“差 1”错误,编译程序可以采用某种方法将其查出,并给出如下的错误信息

```
-> line 23: while (i<=j)
      off by one error: this should be '<'
```

又如,编译程序可以发现算法中有下面的错误:

```
-> line 42: int itoa(int i, char* str)
      algorithm error: itoa fails when i is -32768
```

再如,当出现了参数传递错误时,编译程序可以给出如下的错误信息:

```
-> line 318: strCopy = memcpy(malloc(length), str, length);
      Invalid argument: memcpy fails when malloc returns NULL
```

好了,要求编译程序能够做到这一程度似乎有点过分。但如编译程序真能做到这些,可以想象编写无错程序会变得多么容易。那简直是小事一桩,和当前程序员的一般作法真没法比。

假如在间谍卫星上用摄像机对准某个典型的软件车间,就会看到程序员们正弓着身子趴在键盘上跟踪错误;旁边,测试者正在对刚作出的内部版本发起攻击,轮番轰炸式地输入大量的数据以求找出新的错误。你还会发现,测试员正在检查老版本的错误是否溜进了新版本。可以推想,这种查错方法比用上面的假想编译程序进行查错要花费大得多的工作量、确实如此,而且它还要有点运气。

运气?

是的,运气。测试者之所以能够发现错误,不正是因为他注意到了诸如某个数不对、某个功能没按所期望的方式工作或者程序瘫痪这些现象吗?再看看上面的假想编译程序给出的上述错误:程序虽然有了“差 1”错误,但如果它仍能工作,那么测试者能看得出来吗?就算看得出来,那么另外两个错误呢?

这听起来好象很可怕但测试人员就是这样做的大量给程序输入数据,希望潜在的错误能够亮相。“噢,不!我们测试人员的工作可不这么简单,我们还要使用代码覆盖工具、自动的测试集、随机的“猴”程序、抽点打印或其他什么的”。也许是这样,但还是让我们来看看这些工具究竟做了些什么吧!覆盖分析工具能够指明程序中哪些部分未被测试到,测试人员可以使用这一信息派生出新的测试用例。至于其它的工具无非都是“输入数据、观察结果”这一策略的自动化。

请不要产生误解,我并不是说测试人员的所作所为都是错误的。我只是说利用黑箱方法所能做的只是往程序里填数据,并看它弹出什么。这就好比确定一个人是不是疯子一样。问一些问题,得到回答后进行判断。但这样还是不能确定此人是不是疯子。因为我们没法知道其头脑中在想些什么。你总会这样地问自己:“我问的问题够吗?我问的问题对吗……”。

因此，不要光依赖黑箱测试方法。还应该试着去模仿前面所讲的假想编译程序，来排除运气对程序测试的影响，自动地抓住错误的每个机会。

考虑一下所用的语言

你最后一次看推销字处理程序的广告是什么时候？如果那个广告是麦迪逊大街那伙人写的，它很可能是这么说：“无论是给孩子们的老师写便条还是为下期的《Great American Novel》撰稿，WordSmasher 都能行，毫不费劲！WordSmasher 配备了令人吃惊的 233000 字的拼写字典，足足比同类产品多 51000 个字。它可以方便地找出样稿中的打字错误。赶快到经销商那里去买一份拷贝。WordSmasher 是从圆珠笔问世以来最革命性的书写工具！”

用户经过不断地市场宣传熏陶，差不多都相信拼写字典越大越好，但事实并非如此。象 em、abel 和 si 这些词，在任何一本简装字典中都可以查到、但在 me、able 和 is 如此常见的情况下您还想让拼写检查程序认为 em、abel 和 si 也是拼写正确的词吗？如果是，那么当你看到我写的 suing 时，其本意很可能是与之风马牛不相及的 using。问题不在于 suing 是不是一个真正的词而在于它在此处确实是个错误。

幸运的是，某些质量比较高的拼写检查程序允许用户删去象 em 这类容易引起麻烦的词。这样一来，拼写检查程序就可以把原来合法的单词看成是拼写错误。好的编译程序也应该能够这样——可以把屡次出错的合法的 C 习惯用法看成程序中的错误。例如，这类编译程序能够检查出以下 while 循环错放了一个分号：

```
/* memcpy 复制一个不重叠的内存块 */
void* memcpy(void* pvTo, void* pvFrom, size_t size)
{
    byte* pbTo = (byte*)pvTo;
    byte* pbFrom = (byte*)pvFrom;
    while(size-->0);
        *pbTo++ = *pbFrom++;
    return(pvTo);
}
```

我们从程序的缩进情况就可以知道 while 表达式后由的分号肯定是个错误，但编译程序却认为这是一个完全合法的 while 语句，其循环体为空语句。由于有时需要空语句，有时不需要空语句，所以为了查出不需要的空语句，编译程序常常在遇到空语句时给出一条可选的警告信息，自动警告你可能出了上面的错误。当确定需要用空语句时，你就用。但最好用 NULL 使其明显可见。例如：

```
char* strcpy(char* pchTo, char* pchFrom)
{
    char* pchStart = pchTo;
    while(*pchTo++ = *pchFrom++)
```

```

        NULL;
    Return(pchStart);
}

```

由于 NULL 是个合法的 C 表达式，所以这个程序没有问题。使用 NULL 的更大好处在于编译程序不会为 NULL 语句生成任何的代码，因为 NULL 只是个常量。这样，编译程序接受显式的 NULL 语句，但把隐式空语句自动地当作错误标出。在程序中只允许使用一种形式的空语句，如同为了保持文字的一致性，文中只想使用 zero 的一种复数形式 zeroes，因此要从拼写字典中删除另一种复数形式 zeros。

另一个常见的问题是无意的赋值。C 是一个非常灵活的语言，它允许在任何可以使用表达式的地方使用赋值语句。因此如果用户不够谨慎，这种多余的灵活性就会使你犯错误。例如，以下程序就出现了这种常见的错误：

```

if(ch = ' \t ' )
    ExpandTab();

```

虽然很清楚该程序是要将 ch 与水平制表符作比较，但实际上却成了对 ch 的赋值。对于这种程序，编译程序当然不会产生错误，因为代码是合法的 C。

某些编译程序允许用户在 && 和 || 表达式以及 if、for 和 while 构造的控制表达式中禁止使用简单赋值，这样就可以帮助用户查出这种错误。这种做法的基本依据是用户极有可能在以上五种情况下将等号 == 偶然地键入为赋值号 =。

这种选择项并不妨碍用户作赋值，但是为了避免产生警告信息，用户必须再拿别的值，如零或空字符与赋值结果做显式的比较。因此，对于前面的 strcpy 例子，若循环写成：

```

while(*pchTo++ = *pchFrom++)
    NULL;
编译程序会产生警告信息—所以要写成；
while( (*pchTo++ = *pchFrom++) != ' \0 ' )
    NULL;

```

这样做有两个好处。第一，现代的商用级编译程序不会为这种冗余的比较产生额外的代码，可以将其优化掉。因此，提供这种警告选择项的编译程序是可以信赖的。第二，它可以少冒风险，尽管两种都合法，但这是更安全的用法。

另一类错误可以被归入“参数错误”之列。例如，多年以前，当我正在学 C 语言时，曾经这样调用过 fputc：

```

fprintf(stderr, "Unable to open file %s. \n", filename);
.....
fputc(stderr, ' \n ' );

```

这一程序看起来好象没有问题，但 fputc 的参数次序错了。不知道为什么，我一直认为流指针 (stderr) 总是这类流函数的第一个参数。事实并非如此，所以我常常给这些函数传递过去许多没用的信息。幸好 ANSI C 提供了函数原型，能在编译时自动地查出这些错误。由于 ANSI C 标准要求每个库函数都必须有原型所以在 stdio.h 头文件中能够找到 fputc 的

原型。fputc 的原型是：

```
int fputc(int c, FILE* stream);
```

如果在程序中 include 了 stdio.h, 那么在调用 fputc 时, 编译程序会根据其原型对所传递的每个参数进行比较。如果二者类型不同, 就会产生编译错误。在上面的错误例于中, 因为在 int 的位置上传递了 FILE* 类型的参数, 所以利用原型可以自动地发现前一个 fputc 的错误。

ANSI C 虽然要求标准的库函数必须有原型, 但并不要求用户编写的函数也必须有原型。严格地说, 它们可以有原型, 也可以没有原型。如果用户想要检查出自己程序中的调用错误, 必须自己建立原型, 并随时使其与相应的函数保持一致。

最近我听到程序员在抱怨他们必须对函数的原型进行维护。尤其是刚从传统 C 项目转到 ANSI C 项目时, 这种抱怨更多。这种抱怨是有一定理由的, 但如果不用原型, 就不得不依赖传统的测试方法来查出程序中的调用错误。你可以扪心自问, 究竟哪个更重要, 是减少一些维护工作量, 还是在编译时能够查出错误? 如果你还不满意, 请再考虑一下利用原型可以生成质量更好的代码这一事实。这是因为: ANSI C 标准使得编译程序可以根据原型信息进行相应的优化。

在传统 C 中, 对于不在当前正被编译的文件中的函数, 编译程序基本上得不到关于它的信息。尽管如此, 编译程序仍然必须生成对这些函数的调用, 而且所生成的调用必须奏效。编译程序实现者解决这个问题的办法是使用标准的调用约定。这一方法虽然奏效, 但常常意味着编译程序必须生成额外的代码, 以满足调用约定的要求。但如果使用了“要求所有函数都必须有原型”这一编译程序提供的警告选择项, 由于编译程序了解程序中每个函数的参数情况, 所以可以为不同的函数选择它认为最有效率的调用约定。

空语句、错误的赋值以及原型检查只是许多 C 编译程序提供的选择项中的一小部分内容, 实际上常常还有更多的其它选择项。这里的要点是: 用户可以选择的编译程序警告设施可以就可能的错误向用户发出警告信息, 其工作的方式非常类似于拼写检查程序对可能的拼写错误的处理

Peter Lynch, 据说是 80 年代最好的合股投资公司管理者, 他曾经说过: 投资者与赌徒之间的区别在于投资者利用每一次机会, 无论它是多么小, 去争取利益; 而赌徒则只靠运气。用户应该将这一概念同样应用于编程活动, 选择编译程序的所有可选警告设施, 并把这些措施看成是一种无风险高偿还的程序投资。再不要问: “应该使用这一警告设施吗? 而应该问: “为什么不使用这一警告设施呢?”。要把所有的警告开关都打开, 除非有极好的理由才不这样做。

使用编译程序所有的可选警告设施

增强原型的能力

不幸的是, 如果函数有两个参数的类型相同, 那么即使在调用该函数时互换了这两个参

数的位置，原型也查不出这一调用错误。例如，如果函数 `memchr` 的原型是：

```
void* memchr(const void* pv, int ch, int size);
```

那么在调用该函数时，即使互换其字符 `ch` 和大小 `size` 参数，编译程序也不会发出警告信息。但是如果在相应界面和原型中使用了更加精确的类型，就可以增强原型提供的错误检查能力。例如，如果有了下面的原型：

```
void* memchr(const void* pv, unsigned char ch, size_t size);
```

那么在调用该函数时弄反了其字符 `ch` 和大小 `size` 参数，编译程序就会给出警告错误。

在原型中使用更精确类型的缺陷是常常必须进行参数的显式类型转换，以消除类型不匹配的错误，即使参数的次序正确。

lint 并不那么差

另一种检查错误更详细、更彻底的方法是使用 `lint`，这种方法几乎不费什么事。最初，`lint` 这个工具用来扫描 C 源文件并对源程序中不可移植的代码提出警告。但是现在大多数 `lint` 实用程序已经变得更加严密，它不但可以检查出可移植性问题，而且可以检查出那些虽然可移植并且完全合乎语法但却很可能是错误的特性，上一节那些可疑的错误就属于这一类。

不幸的是，许多程序员至今仍然把 `lint` 看作是一个可移植性的检查程序，认为它只能给出一大堆无关的警告信息。总之，`lint` 得到了不值得麻烦的名声。如果你也是这样想的程序员，那么你也应该重新考虑你的见解。想一想究竟是哪一种工具更加接近于前文所述的假想编译程序，是你正使用的编译程序，还是 `lint`？

实际上，一旦源程序变成了没有 `lint` 错误的形式，继续使其保持这种状态是很容易做到的。只要对所改变的部分运行 `lint`，没有错误之后再把其并入到原版源代码中即可。利用这种方法，并不要进行太多的考虑，只要经过一、二周就可以写出没有 `lint` 错误的代码。在达到这个程度时，就可以得到 `lint` 带来的各种好处了。

使用 lint 来查出编译程序漏掉的错误

但我做的修改很平常

一次在同本书的一个技术评审者共进午餐时，他问我本书是否打算包括一节单元测试方面的内容。我回答说：“不”。因为尽管单元测试也与无错代码的编写有关，但它实际上属于另一个不同的类别，即如何为程序编写测试程序。

他说：“不，你误解了。我的意思是你是否打算指出在将新做的修改并入原版源代码之前，程序员应该实际地进行相应的单元测试。我的小组中的一位程序员就是因为进行了程序的修改之后没有进行相应的单元测试，使一个错误进入到我们的原版源代码中。”

这使我感到很惊奇。因为在 Microsoft，大多数项目负责人都要求程序员在合并修改了

的源代码之前，要进行相应的单元测试。

“你没问他为什么不做单元测试吗？”，我问道。

我的朋友从餐桌上抬起头来对我说：“他说他并没有编写任何新的代码，只是对现有代码进行了某些移动。他说他认为没必要再进行单元测试”。

这种事情在我的小组中也曾经发生过。

它使我想起曾经有一个程序员在进行了修改之后，甚至没有再编译一次就把相应的代码并入了原版源代码中。当然，我发现了这一问题，因为我在对原版源代码进行编译时产生了错误。当我问这个程序员怎么会漏掉这个编译错误，他说：“我做的修改很平常，我认为不会出错”，但他错了。

这些错误本来都应该不会进入原版源代码中，因为二者都可以几乎毫不费力地被查出来。为什么程序员会犯这种错误呢？是他们过高地估计了自己编写正确代码的能力。

有时，似乎可以跳过一些设计用来避免程序出错的步骤，但走捷径之时，就是麻烦将至之日。我怀疑会有许多的程序员甚至没有对相应的代码进行编译，就“完成”了某一功能。我知道这只是偶然情况，但绕过单元测试的趋势正在变强，尤其是作简单的改动。

如果你发现自己正打算绕过某个步骤。而它恰恰可以很容易地用来查错，那么一定要阻止自己绕过。相反，要利用所能得到的每个工具进行查错。此外，单元测试虽然意味着查错，但如果你根本就不进行单元测试也是枉然。

如果有单元测试，就进行单元测试

小结

你认识哪个程序员宁愿花费时间去跟踪排错，而不是编写新的代码？我肯定有这种程序员，但至今我还没有见过一个。对于我认识的程序员，如果你答应他们再不用跟踪下一个错误，他们会宁愿一辈子放弃享用中国菜。

当你写程序时，要在心中时刻牢记着假想编译程序这一概念，这样就可以毫不费力或者只费很少的力气利用每个机会抓住错误。要考虑编译程序产生的错误、lint 产生的错误以及单元测试失败的原因。虽然使用这些工具要涉及到很多的特殊技术，但如果不花这么多的功夫，那产品中会有多少个错误？

如果想要快速容易地发现错误，就要利用工具的相应特性对错误进行定位。错误定位得越早，就能够越早地投身于更有兴趣的工作。

要点：

- 消除程序错误的最好方法是尽可能早、尽可能容易地发现错误，要寻求费力最小的自动查错方法。
- 努力减少程序员查错所需的技巧。可以选择的编译程序或 lint 警告设施并不要求程序员要有什么查错的技巧。在另一个极端，高级的编码方法虽然可以查出或减少

错误，但它们也要求程序员要有较多的技巧，因为程序员必须学习这些高级的编码方法。

练习：

- 1) 假如使用了禁止在 while 的条件部分进行赋值的编译程序选择项，为什么可以查出下面代码中的运算符优先级错误？

```
While(ch = getchar() != EOF)
```

.....

- 2) 看看你怎样使用编译程序查出无意使用的空语句和赋值语句。值得推荐的办法是进行相应的选择，使编译程序能够对下列常见问题产生警告信息。怎样才能消除这些警告信息呢？

a) `if(flight == 063)`。这里程序员的本意是对 63 号航班进行测试，但因为前面多了一个 0 使 063 成了八进制数。结果变成对 51 号航班进行测试。

b) `If(pb != NULL & pb != 0xff)`。这里不小心把 `&&` 键入为 `&`，结果即使 `pb` 等于 `NULL` 还会执行 `*pb != 0xff`。

c) `quot = numer/*pdenom`。这里无意间多了个 `*` 号结果使 `/*` 被解释为注释的开始。

d) `word = bHigh<<8 + bLow`。由于出现了运算符优先级错误，该语句被解释成了：
`word = bHigh << (8+bLow)`

- 3) 编译程序怎样才能对“没有与之配对的 `else`”这一错误给出警告？用户怎样消除这一警告？

- 4) 再看一次下面的代码：

```
if(ch == '\t')
```

```
    ExpandTab();
```

除禁止在 `if` 语句中使用简单赋值的方法之外，能够查出这个错误的另一种众所周知的方法是把赋值号两边的操作数颠倒过来：

```
if( '\t' == ch)
```

```
    ExpandTab();
```

这样如果应该键入 `==` 时键入了 `=`，编译程序就会报错，因为不允许对常量进行赋值。这个办法彻底吗？为什么它不象编译程序开关自动化程度那么高？为什么新程序员会用赋值号代替等号？

- 5) C 的预处理程序也可能引起某些意想不到的结果。例如，宏 `UINT_MAX` 定义在 `limit.h` 中，但假如在程序中忘了 `include` 这个头文件，下面的伪指令就会无声无息地失败，

因为预处理程序会把预定义的 `UINT_MAX` 替换成 0：

.....

```
#if UINT_MAX > 65535u
```

.....

```
#endif
```

怎样使预处理程序报告出这一错误？

课题：

为了减轻维护原型的工作量，某些编译程序会在编译时自动地为所编译的程序生成原型。如果你用的编译程序没有提供这一选择项，自己写一个使用程序来完成这一工作。为什么标准的编码约定可以使这个使用程序的编写变得相对容易？

课题：

如果你用的编译程序还不支持本章（包括练习）中提及的警告设施，那么促进相应的制造商支持这些设施，另外要敦促他们除了允许用户设定或者取消对某些类错误的检查之外，还要提供有选择地设定或取消一些特定的警告设施，为什么要这样做呢？