Visual C++

除错手册

【美】Keith Bugg 著

怀石工作室
常小刚 译

中国电力出版社
www.cepp.com.cn

R&D BOOKS

Visual C++ 除错手册

Visual C++

除错手册

【美】Keith Bugg 著

怀石工作室
常小刚 译

中国电力出版社

内 容 提 要

本书讲述了 Windows 程序开发中至关重要却往往易于忽视的一个环节——调试。本书言简意赅，以最为流行的开发工具 Visual C++为例，讲述调试中所必备的知识以及应对方法、常用工具，充满了实用的技巧和宝贵的实践经验，是程序员必备的工具书。

本书适合各种级别的软件开发人员阅读，也可用作高校计算机相关专业编程课的参考读物。

图书在版编目（CIP）数据

Visual C++ 除错手册/ （美）巴格（Bugg, K.）著；常小刚译. -北京：中国电力出版社，2000.6

ISBN 7-5083-0357-1

Ⅰ.V… Ⅱ.①巴…②常… Ⅲ.C 语言-程序设计 Ⅳ.TP312

中国版本图书馆 CIP 数据核字（2000）第 32178 号

本书由美国 Miller Freeman, Inc.公司授权出版

JS432/26

中国电力出版社出版、发行

（北京三里河路 6 号 100044 http://www.cepp.com.cn）

实验小学印刷厂印刷

各地新华书店经售

*

2000 年 6 月第一版 2000 年 6 月北京第一次印刷

787 毫米×1092 毫米 16 开本 9.75 印张 214 千字

定价 19.00 元

前　言

本书讲述了 Windows™ 程序软件开发生命周期中最为至关重要却往往不受重视的一个阶段——调试。然而，本书并不只是简单地解释如何使用调试器，或者严格地从理论上讨论你可能会遇到的错误类型。更准确地说，它还希望让你更好地理解错误是如何被引入到程序中的，你如何通过适当的设计技术来尽量减少它们，以及你如何能够检测并修正它们。但不要担心；这里仍然是有一些技巧的。我想把这本书定位在初级到中级的 Windows 开发者——也就是说，我假定你熟悉 Visual C++ Workbench，知道如何使用它。在本书中，我认为读者有足够的知识，不需要调试上的帮助，而是想要更进一步地学习。换句话说，你或许已经是，或者正在成为一名相当好的软件开发人员。

本书是针对 Microsoft Visual C++开发工具的，这是一个现在最常用的平台以及在主流的 Windows 开发中占统治地位的工具。本书的组织如下：

- ✧ 错误的分类。
- ✧ 在设计和分析过程中将错误尽量减少。
- ✧ 内存和内存分配。
- ✧ 使用 Visual C++调试器。
- ✧ 使用商业的调试器。

当你读完本书后，你会了解到许多种错误和它们的起因，以及建议的解决方案。我对编译器中已知的错误会给予相应的注意，而如果你陷入困境的话，你能够在最后一章找到一个可能会对你有帮助的资源纲要。

除了关于调试的主题，你还会看到一些有用的提示、技巧和工具来辅助进行基本的工程管理。虽然其中有一些话题同错误的检测和解决没有直接的联系，但它们能够间接地影响你的调试过程。

例如，在第四章中有一个关于 pragma 的讨论。其中一个有用的 pragma 为：

```
#message ( string )
```

它使你能够往构建窗口中输出一个消息。比如：

```
#message ("Date-validation not finished in method GetDate() ! ");
```

如果你有一个函数还没有完全完成，这样一个提示能够减少一些障碍，节省你提交产品的时间。

最后，我希望有了本书的帮助，你可以尽快将自己基于 MFC/Visual C++的应用程序推向市场，而同时又尽可能地减少程序的错误（bug）。

目　录

第一章

简介和范围

有过一段时间 Windows 应用程序开发经历的人都能体会到，编写代码只是程序开发的一半，另一半工作则是让代码正确的运行。实际上，软件的开发过程要经历三个步骤：

1. 让程序能够运行。
2. 让程序能够正确运行。
3. 让程序能够高效率地运行。

本书将紧紧围绕如何使程序正确运行讲述——开发者应该具备哪些知识；如何顺利地实现自己的想法；如何按自己的意愿使用各种工具。由于现在 Microsoft Visual C++已逐渐成为开发 Windows 程序事实上的标准，所以本书就以 Microsoft Visual C++为平台。

本书的目的不仅仅是给读者介绍调试程序的过程，而且还要尽可能地让读者了解程序中各种错误（bug）产生的根源。例如，程序中产生错误可能是由于对涉及目标的理解不够深入或前后矛盾，也可能是由于测试过程不完善或不完全（特别是那些面向多平台的软件），等等。合适的调试策略、调试工具和调试技术，是软件开发过程中非常重要的环节，因为错误将消耗大量的时间和财力，并对软件当前和未来的销售产生极大的影响。虽然人们已经开始接受，大型、复杂的软件产品不可避免地会有一些错误，但是如果错误的数量和程度超过人们所能容忍的限度，恐怕就会给软件新版本和新产品的销售带来困难。

1.1 事件驱动的范例

Windows 应用程序是基于事件或消息模型的，这使得调试 Windows 程序非常复杂。与以线性模式运行，基于过程调用的程序不同，Windows 的运行是对消息做出响应。谁也无法预料接下来会产生什么消息，你也就无从知道应该去检查哪里。所以，要调试基于事件的应用程序，开发者一定要掌握几种基本的调试工具。Visual C++的专业版就提供了许多工具，包含了全面的分析错误功能。忘记释放为 Cbrush 对象所申请的内存了？那也不用担心——STRESS.EXE 返回的内存分配信息会告诉你 GDI 内存堆正在减少。这至少会给

你指出一个解决问题的正确方向。如果你不去用，或者根本不懂得如何去用这些工具，那么调试 Windows 程序无疑是难上加难。

1.2 迎战错误

怎样消灭程序中的错误，可以从三个不同的方面来看：

1. 预防错误。
2. 检测错误。
3. 解决或处理错误。

通过合适的软件分析和设计策略来预防错误，这一点比较容易——这些错误就永远不会进入系统中。而接下来的两个工作——这也正是这本书的主题——就要借助 Visual C++ 提供的调试工具，或者你能够从第三方买到的工具。通过使用这些工具，并采用可靠的、系统的测试方法，通常就能很好的检测和排除错误。调试的过程是通过检查源代码来检测程序中的某些不规则之处，而不管它们是否是真正的错误。无论如何，提供给用户一个完善的从错误恢复的路径，这总该是一个好的设计原则。

1.3 错误源和错误分类

在软件开发周期中，错误可能出现在任何时间、任何地点，开发者得了解这些错误，并把它们分类以易于处理。机器错误（马上我就会谈到）是一个典型的例子——如果你不了解它，那么借助任何一种监视窗口和调试器， 也不能解释为何一次计算会得出明显是错误的结果来。

我们把开发周期中经常出现的主要错误分为以下四个类别：

1. 机器错。
2. 编译错。
3. 运行错。
4. 逻辑和设计错误。

第一种类型，也就是机器错，在大多数开发者看来似乎并不理解或者不重视。接下来的两种错误，编译错和运行错，其大部分是易于检测和排除的。而最后一种错误，设计逻辑错，则非常的隐蔽，因为你的程序能够运行，只是会产生错误的或者不可预知的结果。没有任何一种调试器能够解决这个问题；只有细致、可靠的分析设计策略，才能帮你避免这种错误。

机器错

尽管主流文化不断吹嘘计算机的精度，但实际并不是那样。那些小小的硅片上不可避免的会有错误——有些数字无法用计算机精确表达。最常见的一种情况叫做**舍入错误**。简单的说，舍入错误产生的原因是计算机不把一个二进制数的所有位都保存。在程序中使用数字变量时的，它们的值都会被转换为二进制。CPU 能够存取的二进制位数是有限的，它会把多出来的数位舍弃，这样得到的结果就不精确了。类似这样的情况平时也会遇到，我们日常计算用的是十进制。如果用 6.0 除 10，得到的结果是 1.66666……，我们通常也会把末尾的 6 都舍去。

现在来看一个实例，请看下面的代码：

```
float      fSum = 0.0;
for (int i = 0; i<1000000; i++)
      fSum += .000001;
```

把 1/100,000 重复加 100,000 次，结果应该恰好是 1。然而由于机器错误，fSum 的值往往会比 1 稍大或稍小。我用来测试的机器是奔腾 166MHz，最后 fSum 的值是 1.009039。如果把 fSum 的类型从 float 改为 double，那就会得到正确的结果。但在精度要求非常严格的数字计算中，这样就难以得到精确的结果了。计算机并不善于处理浮点数，也处理不好把比较大的数加到比较小的数上这种情况。这些操作都会增加程序出错的机率，有时甚至会得出奇怪的或无法预料的结果。受机器错误影响最大的，是一些对计算精度要求非常高的环境，比如气象模型计算，或是最小二乘法曲线模拟，等等。

编译错

这种错误是最容易发现和排除的——编译器就会替你完成全部工作。下面是一个例子。

```
Cstring csStr;
Int len = csStr.GetLengthh（）;
```

如果编译这段代码，你就会得到如下的错误信息：

```
error C2039: ' GetLengthh': is not a member of 'Cstring'
```

这里有一个简单的拼写错误——类的方法 GetLength（）被拼错了，所以编译器不认识。后面还会讲到，你可以通过改变 Visual C++编译器的设置，来辅助进行错误检测和修正。

运行错

运行错发生在执行程序的时候。其起因或者是、或者不是逻辑错误，总之程序无法运行。程序编译时没有遇到任何错误和警告信息，但会在运行过程中产生意外。下面是一个运行错的简单例子。

```
int k = 0，i = 100;
for （int j = 9; j > 1; j--）
      k = i / j;
```

这一小段代码编译时没有任何错误；执行时也正好循环九次。但在最后一次循环中，会把变量 i 用 0 除，这就产生了运行错。实际上，被零除更准确的说应该是**硬件例外**。尽管在开发者看来这二者都差不多，但检测和解决例外的方式与错误仍有所不同。例外有两种：硬件例外和软件例外。这在下一章中会更详细地分析。

最容易导致运行错误的一个原因就是使用的变量没有初始化，看下面一个例子。

```
For (int j; j<10; j++)
{
    //do somthing
}
```

这段代码中，按照作者的意图 j 的初始值应该是 0，但却没有对变量 j 进行初始化。如果编译器用来存储变量 j 的地址原先有一个非零的值，那这个循环就可能会运行不正常。而如果在程序中有一个指针没有初始化的话，就可能引起非常严重的后果。

运行错也可能是由编译器本身的错误引起的。后面我们还会详细讲到，在这里先给一个小例子。

```
Int i, j =0;
For (int i=0; i<5; i++)
{
    j = (j<3) ? j++: 0;        //j never incremented!
}
```

这段代码是完全正确的。然而由于 Visual C++编译器中的一个错误，变量 j 并不会像所期望的那样增长三次。尽管编译器错误引起的问题非常之少，但这种错误确实存在，而且当程序运行有误时，应当考虑到这种原因。

逻辑和设计错误

在前面的分类中，逻辑和设计错误是相当难以纠正的。你要清楚正在开发的软件可能会包含互相矛盾的需求和设计目标。例如你正在为客户开发一个进货控制软件包。其中财务管理部门要求有一份这个月的已收货物的清单，以根据这份清单为本月内接收的货物付

款。而另一个部门又要先检查所有的进货以确保没有遗失和损坏。如果检查部门持有一份货物超过一个月，而你的软件又没有任何措施处理这种情况，财务部门就无法及时付款，这会影响客户的信誉。在这种情况下，进行系统分析的人是否理解客户方的事务规则，是一个关系到客户命运的大事。

根据我的经验，许多逻辑错误都是由“不完整的 if 语句”引起的——也就是说，开发者写了一个 if 语句，却没有相应的 else 语句。这有时会严重违背了客户方的秩序，使情况变得复杂。程序的作者往往是把注意力放在了一种情况的处理上，却忽视了与之相对的另一种情况（也就是 else 语句）。一个例子是日期处理。你想让你的代码区别闰年和非闰年，而你判断的方法是“这个年份如果能被 4 整除，那它就是闰年”。但这并不完全——你还需要检查这个年份是否能被 400 整除。有时非常微小的忽视却可能会导致严重的后果。

德·摩根定律

逻辑错误也可能是由于编程时没有遵守德·摩根定律所引起的。这个定律是关于两个布尔变量（或布尔表达式）取反后进行运算的定律。德·摩根定律的两种基本情况用逻辑符号表示如下：

```
!p AND !q = !p OR !q
!p OR !q = !p AND !q
```

通常，德·摩根定律意味着，如果你想处理的是“不满足条件一的情况以及(*AND*)不满足条件二的情况”，那么用布尔语句表述时一般就应该用 OR 运算符(!p *AND* !q = !p OR !q)；如果你想处理的是“不满足条件一或者（*OR*）条件二的情况”，那么用布尔语句表述时一般就应该用 AND 运算符(!p *OR* !q = !p AND !q)；我们看下面这个 C 的例子。

```
Int x = 1;
Int y = 2;
If (!(x ==1) && !(y ==3))
    Printf("The right answer!");
Else
    Printf("The wrong answer! ");
```

尽管 x 确实等于 1，而 y 也确实不等于 3，但运行这段代码后会打印出“The wrong answer!”。根据德·摩根定律，代码中的表达式应该改成下面这个样子：

```
if (!((x == 1) && (y == 3)))
    printf("The right answer!");
else
    printf("The wrong answer!");
```

现在运行后会打印出“The right answer!”来了。

现在来看一个更实际的例子。假如你的程序是监测一个制药车间的：在车间内的温度不高于 150 度，以及（*AND*）气压不低于 3 个大气压这两种情况下，这个车间都是不安全的。看下面的代码：

```
int temp = 175;                    //assume analog-to-digital sensors report
int pressure = 5;                  //these conditions,which make the plant unsafe
if (!(temp>150)&&!(pressure <3))
    printAlarm("The plant is unsafe!\n");
else
    printAlarm("The plant is safe!\n");
```

在这里，表达式的前一部分是 TRUE——温度是高于 150 度的，但后一部分就是 FALSE——气压是高于 3 个大气压的。如果你用调试器跟踪这段代码的话，你会发现实际上执行了 else 语句，结果报告车间是安全的，但实际上车间并不安全。再一次应用德·摩根定律，正确的代码如下：

```
if (!(temp > 150) || !(pressure < 3))
    printAlarm(" The plant is unsafe!\n" );
else
    printAlarm(" The plant is dafe.\n" );
```

这是一种很容易就会犯的错误，而且无法借助调试器发现这种错误。所以编写代码时要养成良好的习惯，对于包含两个“非”运算的混合逻辑表达式一定要仔细检查。

1.4 最小化错误的数量

要想写出一个可靠、无错的程序，在一开始就要设法减少各种出错的可能性。从具体手段上来看，有一些很普通的方法可以帮助你减少错误，或者至少是让错误更容易被发现。这一节里，我介绍一些编程时值得养成的习惯，以利于发现程序中隐藏的错误。

命名的习惯和注释的风格

程序员应当在遵循匈牙利命名法的基础之上，形成一个良好的命名习惯。例如，为 CString 类型的变量名命名的时候，除了添加前缀“cs”之外，最好能包含关于这个变量功能的详细信息。举个例子，变量名 csNameRemoteDept 包含的信息就比 csDept 要多许多。首先从这个变量名中就不仅可知这是一个部门名，而且是一个远程的部门。在你首次看到这个符号时你可能就会猜想代码中还有另一个变量名为 csNameLocalDept，或者类似的东西。在代码中要尽量用这种有意义的符号。比如说在这个例子中，程序员所处理的可能就是远程部门的而不是当地的部门。每个程序员都会形成自己的命名习惯。在编程时一定要

仔细检查，或者请同事或朋友给你提意见。

个性化你的工作空间

不知你有没有发现，只要做得好的话，定制 Visual C++ workbench 有助于你在编程时预防错误。例如，你可以让变量名在源代码窗口（或者其他窗口）中以醒目的彩色显示。这样从视觉上就能够迅速轻易地定位，从而有助于发现和预防错误。

要设置颜色的话，启动 VC++，在菜单栏上“Tools”下面选择“Options...”，在弹出的对话框中单击“Format”标签，此时对话框如图 1-1 所示。

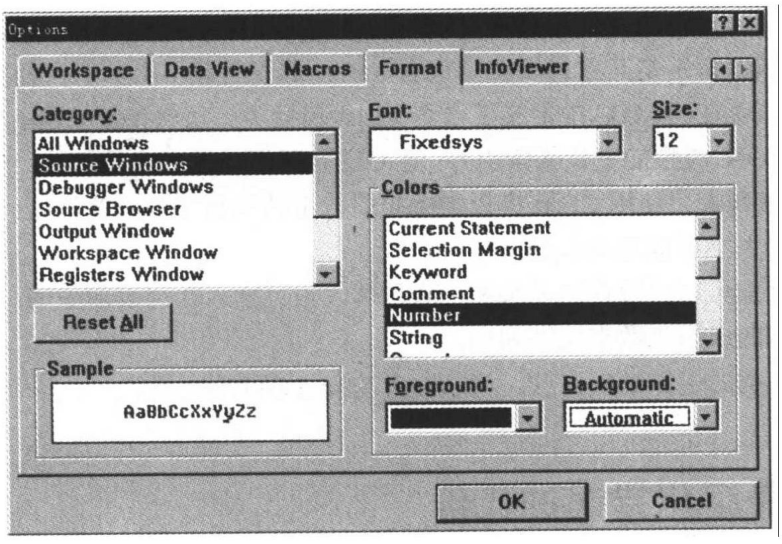

图1-1 格式选项对话框

在“Category”列表框中选择“Source Windows”，滚动“Colors”列表框，在其中选择想以彩色显示的符号类型（比如说变量名），然后选择前景颜色和背景颜色。要注意的是，不要设置过多的符号用彩色显示，不然会令你眼花缭乱。从图 1-1 中我们也可以看到，还可以设置符号显示时的字体和大小，更适合自己的爱好。总之，“个性化”就是要把你的工作空间设置的舒适协调——你将会在那儿工作。

数组引起的错误

如果要使用数组，那就得对数组的索引非常了解，并且使用时也得十分小心。在基于 MFC 或者说是通过 Visual C++建立的程序中，使用数组很可能会造成隐患。MFC 是用 C++编写的，而 C++其实无非就是 C 的一个超集。大家都知道，严格地说 C 中是没有数组的。比如说如果你声明了一个 10 个整数的数组，那么编译器会分配一个地址和从这个地址开始的 10 个整数所占用的内存空间。这样还不会有什么问题，问题是在运行时 C 中没有数组索引的边界检查，所以程序中对数组的索引可以是任意的！例如：

```
int iAges[10];                    //declare an array of 10 integers
int k = 0, index = 0;
//
//    do something else here...
//
k = iAges[index];                 //no problem UNLESS index becomes > 9
```

在这个简化了的例子中，如果在程序的其他地方改变了 index，导致 index 比 9 大，那变量 k 就会包含一个无意义的值，因为由 iAges[index]所指向的地址中可能存储的是任何值。这并不是说不要使用数组，只是对数组下标的操作必须小心。对于小的，普通的数据成分，比如说一年中的月份，总共只有 12 个，用数组不会有什么问题，但对于范围比较大的数据领域，特别是其数目没有确定上限的，那就应该不用数组，而用其他的结构代替，比如列表。现在的 MFC 中定义了一个 CList 模板类。虽然使用 CList 比较麻烦（还得用一个 POSITION 类型的变量来索引），但 CList 提供了很多成员函数，借助这些函数能够很方便的操纵列表。使用 CList 模板类比起用数组结构有下面一些好处：

- ✧ 列表的大小可以动态改变。
- ✧ 新的元素可以放置在列表头和列表尾端，也可插入在列表的中间。
- ✧ 有丰富的成员函数来操纵列表。

关于 CList 类的更详细的信息，可以参考 Visual C++的文档。

1.5 最小化错误的代价

要使程序可靠无错，最有效的做法是预防错误，或者在开发过程中尽早发现并修正错误。这些问题同数据完整性、系统测试以及程序员编码技巧等紧密相关。这一节探究一下预防错误、或者至少把错误控制到最小程度的技术和过程。

🕮 设计、分析和评论

要控制错误，与项目有关的人都必须懂得开发软件的一般过程。良好的项目管理、同客户的交流与反馈，以及清楚明了的规格说明书，都十分有助于开发出健壮无错的软件产品。不幸的是，上层的管理人员在这一点上往往认识比较肤浅——他们认为交流是在浪费时间，这些时间都应该用来编写代码。就像老话说的那样“There is never enough time to do it right,but there is always enough time to do it over”。对这种短视观点最好的抵御是强烈的个性。项目的组织者应该负起这个责任，保证把设计和分析工作做好，这需要机智、魅力和自信。同时，使用一些标准化的技术，诸如数据库规格化或者软件设计标准等，也有助于确保正确理解项目设计的目标，保证解决方案没有纰漏。

图 1-2 展示了在软件开发的不同阶段修正错误所花代价的对比。从图中可以看到，在

开发的早期修正错误代价要小得多。

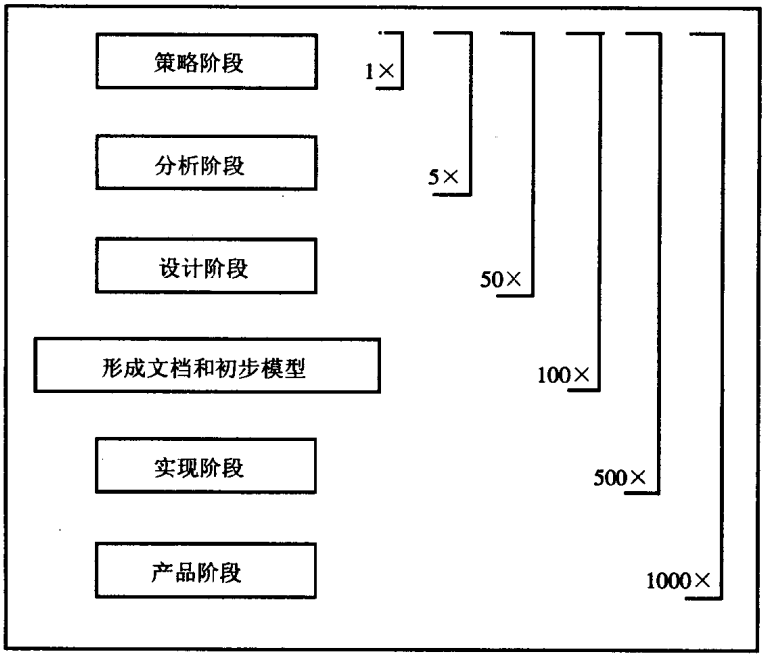

图1-2　软件开发过程中不同阶段修正错误的代价对比

绘制草图

在设计阶段的早期可以做出一张大致的草图给客户看。草图未必就是项目被完成的样子，它只是一种反馈的样式，来确认项目需求是否被很好的理解了，以及客户对这种初步的设想是否满意。随着客户那里相关信息收集得越来越多，相应的也要不时对草图做出修正。

使用草图反馈技术也有一些缺陷需要注意。有一个缺陷就是，草图会导致双方把焦点放在一些次要的细节问题上。例如，在你的草图中有一个对话框，其中使用了复选框（Check Box），而客户却希望使用单选按钮（Radio Button），千万不要让双方的会面变成一场喜欢单选按钮还是复选框的争论。这只是次要的细节——更重要的目的是确认这个对话框的功能设计是否合理。

还有一个缺陷就是客户往往会对项目进度产生误解。此时项目的负责人就需要适当施加压力，提醒客户草图的意义和目的。要通过某种方式努力降低客户的过高期望。适时地提醒客户你们还只是处于项目的分析设计阶段，现在适当地忍耐，回报将是一个很好的符合甚至超过他们要求的软件工具。要指出在项目前期“损失”的时间都会在后期得到弥补，因为那时需要修复的错误就要少得多，设计变成了实现，等等。要记住，代表客户方利益来跟你谈判的人跟你一样，是对其他人负责的，所以一定和对方密切合作，双方共同应付来自高层的压力，这样才会让双方最终都得到满意的结果。