vs中的程序有debug和release两个版本，Debug通常称为调试版本，通过一系列编译选项的配合，编译的结果通常包含调试信息，而且不做任何优化，以为开发 人员提供强大的应用程序调试能力。而Release通常称为发布版本，是为用户使用的，一般客户不允许在发布版本上进行调试。所以不保存调试信 息，同时，它往往进行了各种优化，以期达到代码最小和速度最优。为用户的使用提供便利。

       debug程序通常比release程序要慢，尤其是处理视频方便release要比debug快很多。在release模式对程序进行调试的时候经常会遇到变量虽然初始化了，但是在查看其值的时候却发现是一个随机的数并不是初始化的值，有时候在对变量进行监视的时候了，会出现找不到变量的情况，原因大致如下：

     debug跟release在初始化变量时所做的操作是不同的，debug是将每个字节位都赋成0xcc， 而release的赋值近似于随机。如果你的程序中的某个变量没被初始化就被引用，就很有可能出现异常：用作控制变量将导致流程导向不一致；用作数组下标将会使程序崩溃；更加可能是造成其他变量的不准确而引起其他的错误。所以在声明变量后马上对其初始化一个默认的值是最简单有效的办法，否则项目大了你找都没地方找。代码存在错误在debug方式下可能会忽略而不被察觉到。debug方式下数组越界也大多不会出错，在release中就暴露出来了，这个找起来就比较难了。

   只有DEBUG版的程序才能设置断点、单步执行、使用 TRACE/ASSERT等调试输出语句。REALEASE不包含任何调试信息，所以体积小、运行速度快。

主要参考：

<http://javafans.iteye.com/blog/609937/>

**Debug与Release版本的区别**   
  
    Debug 和 Release 并没有本质的区别，他们只是VC预定义提供的两组编译选项的集合，编译器只是按照预定的选项行动。如果我们愿意，我们完全可以把Debug和Release的行为完全颠倒过来。当然也可以提供其他的模式，例如自己定义一组编译选项，然后命名为MY\_ABC等。习惯上，我们仍然更愿意使用VC已经定义好的名称。   
    Debug版本包括调试信息，所以要比Release版本大很多（可能大数百K至数M）。至于是否需要DLL支持，主要看你采用的编译选项。如果是基于 ATL的，则Debug和Release版本对DLL的要求差不多。如果采用的编译选项为使用MFC动态库，则需要MFC42D.DLL等库支持，而 Release版本需要MFC42.DLL支持。Release不对源代码进行调试，不考虑MFC的诊断宏，使用的是 MFC  Release库，编译时对应用程序的速度进行优化，而Debug则正好相反，它允许对源代码进行调试，可以定义和使用MFC的 诊断宏，采用MFC  Debug库，对速度没有优化。      
    既然Debug和 Release仅仅是编译选项的不同，那么为什么要区分Debug和Release版本呢？   
    Debug和Release，在我看来主要是针对其面 向的目标不同的而进行区分的。Debug通常称为调试版本，通过一系列编译选项的配合，编译的结果通常包含调试信息，而且不做任何优化，以为开发 人员提供强大的应用程序调试能力。而Release通常称为发布版本，是为用户使用的，一般客户不允许在发布版本上进行调试。所以不保存调试信 息，同时，它往往进行了各种优化，以期达到代码最小和速度最优。为用户的使用提供便利。   
  
    下面仅就默认的Debug和Release版本的选项进行 比较，详细的编译选项可以看MSDN的说明。   
我们将默认的Debug和Release的选项设置进行比较，过滤掉相同设置，主要的不同如下：   
编译选项：/Od /D "\_DEBUG" /Gm /RTC1 /MDd /Fo"Debug““" /ZI   
链接选项：/OUT："D：“MyProject“logging“Debug“OptionTest.dll" /INCREMENTAL   
Release设置：   
编译选项：/O2 /GL /D "NDEBUG" /FD /MD /Fo"Release““" /Zi   
链 接选项：/OUT："D：“MyProject“logging“Release“OptionTest.dll" /INCREMENTAL：NO   
  
Debug  版本：    
/MDd  /MLd  或  /MTd  使用  Debug  runtime  library(调试版本的运行 时刻函数库)    
/Od  关闭优化开关    
/D  "\_DEBUG"  相当于  #define  \_DEBUG,打开编译调试代码 开关(主要针对assert函数)    
/ZI  创建  Edit  and  continue数据库，在调试 过程中如果修改了源代码不需重新编译    
/GZ  可以帮助捕获内存错误    
/Gm  打开最小化重链接开关，减少链接时 间    
Release  版本：      
/MD  /ML  或  /MT  使用发布版本的运行时刻函数库    
/O1  或  /O2  优 化开关，使程序最小或最快    
/D  "NDEBUG"  关闭条件编译调试代码开关(即不编译assert函数)    
/GF  合并重 复的字符串，并将字符串常量放到只读内存，防止被修改    
  
MDd与MD   
    首 先，Debug版本使用调试版本的运行时库（/MDd选项），Relase版本则使用的是发布版本的运行时库（vcrt.dll）。其区别主要在于运行时 的性能影响。调试版本的运行时库包含了调试信息，并采用了一些保护机制以帮助发现错误，也因此，其性能不如发布版本。编译器提供的Runtime Library很稳定，不会造成Release版本错误，倒是由于Debug版本的Runtime Library加强了对错误的检测，如堆内存分配检查等，反而会报告错误，应当指出，如果Debug有错误，而Release版本正常，程序肯定是有 Bug的，只是我们还没有发现。   
ZI与Zi   
    其次，/ZI选项与/Zi选项。通过使用/ZI选项，可以在调试过程修改代码 而不需要重新编译。这是个调试的好帮手，可如果我们使用Release版本，这将变得不可行。   
  
Od与O2   
/O2与/Od 选项：Od是关闭编译器优化，普遍用于Debug版本。而O2选项是创建最快速代码，这当然是Release版本的不二选择。   
  
RTCx选 项   
/RTCx选项让编译器插入动态检测代码以帮助你检测程序中的错误。比如，它会将局部变量初始化为非零值。包括用 0xCC初始化所有自动变量，0xCD初始化堆中分配的内存（即new的内存），使用0xDD填充被释放的内存（即delete的内存），0xFD初始化 受保护的内存（debug版在动态分配内存的前后加入保护内存以防止越界访问）。这样做的好处是这些值都很大，一般不可能作为指针，考试，大提示作为数值 也很少用到，而且这些值很容易辩认，因此有利于在Debug版本中发现Release版才会遇到的错误。另外，通过函数指针调用函数时，会通过检 查栈指针验证函数调用的匹配性（防止原型不匹配）。使用/RTCx选项会造成Debug版本出错，而Release版本正常的现象，因为 Release版中未初始化的变量是随机的，很可能使指针指向了有效但是错误的地址，从而掩盖了错误。这个编译选项只能在/Od选项下使用。   
  
Gm，INCREMENTAL or NO   
编译选项中的Gm和链接选项中的 INCREMENTAL都只为一个目的，加快编译速度。我们经常遇上这样的问题，只修改了一个头文件，结果却造成所有动态库的重新编译。而这两个选项就是 为了解决这样的问题。如果启用了/Gm开关，编译器在项目中的.idb文件中存储了源文件和类定义之间的依赖关系。之后的编译过程中使用.idb 文件中的信息确定是否需要编译某个源文件，哪怕是此源文件已经包含了已修改的.h文件。   
INCREMENTAL开关默认是开启的。使用增量链接生 成的可执行文件或者动态链接库会大于非增量链接的程序，因为有代码和数据的填充。另外，增量链接的文件还包含跳转trunk以处理函数重定位到新地址。   
MSDN 上明确指出：为确保最终发布版本不包含填充或者trunk，请非增量链接程序。   
  
/GZ  选项：做以下这些事    
1.初 始化内存和变量。包括用  0xCC  初始化所有自动变量，0xCD  (  Cleared  Data  )  初始化堆中分配的内存（即动态分配 的内存，例如  new  ），0xDD  (  Dead  Data  )  填充已被释放的堆内存（例 如  delete  ），0xFD(  deFencde  Data  )  初始化受保护的内存（debug  版在动态分配内存的前后加入保护内 存以防止越界访问），其中括号中的词是微软建议的助记词。这样做的好处是这些值都很大，作为指针是不可能的（而且  32  位系统中指针很少是奇数值， 在有些系统中奇数的指针会产生运行时错误），作为数值也很少遇到，而且这些值也很容易辨认，因此这很有利于在  Debug  版中发 现  Release  版才会遇到的错误。要特别注意的是，很多人认为编译器会用  0  来初始化变量，这是错误的（而且这样很不利于查找错 误）。    
2.   通过函数指针调用函数时，会通过检查栈指针验证函数调用的匹配性。（防止原形不匹配）    
3.   函数返回前检查 栈指针，确认未被修改。（防止越界访问和原形不匹配，与第二项合在一起可大致模拟帧指针省略  FPO  ）    
  
    通常  /GZ  选 项会造成  Debug  版出错而  Release  版正常的现象，因为  Release  版中未初始化的变量是随机的，这有可能使指针指向一 个有效地址而掩盖了非法访问。    
  
\_DEBUG与NDEBUG   
这是最重要的一个选项。这两个是编译器的预处理器定义，默认情况下\_DEBUG用于Debug版本，而NDEBUG用于Release版本。 它们可以说是重要的无以复加。因为，assert系列的断言仅仅在\_DEBUG下生效！   
下面是assert.h文件中摘出来的：

**C++代码  [收藏代码](javascript:void())**

1. #ifdef NDEBUG
2. #define assert(\_Expression)     ((void)0)
3. #else  /\* NDEBUG \*/
4. #ifdef \_\_cplusplus
5. **extern** "C" {
6. #endif  /\* \_\_cplusplus \*/
7. \_CRTIMP **void** \_\_cdecl \_wassert(\_\_in\_z **const** **wchar\_t** \* \_Message, \_\_in\_z **const** **wchar\_t** \*\_File, \_\_in unsigned \_Line);
8. #ifdef \_\_cplusplus
9. }
10. #endif  /\* \_\_cplusplus \*/
11. #define assert(\_Expression) (void)( (!!(\_Expression)) || (\_wassert(\_CRT\_WIDE(#\_Expression), \_CRT\_WIDE(\_\_FILE\_\_), \_\_LINE\_\_), 0) )
12. #endif  /\* NDEBUG \*/

可以看出在未定义\_DEBUG时，assert变成一条空语句不被执行。   
也就是说，我们现在所有发布的版本无法使 用断言机制进行程序调试。   
  
相关经验：   
1. 变量。   
    大家都知道，debug跟release在初始化变量时所做的操作是不同的，debug是将每个字节位都赋成0xcc， 而release的赋值近似于随机。如果你的程序中的某个变量没被初始化就被引用，就很有可能出现异常：用作控制变量将导致流程导向不一致；用作数组下标将会使程序崩溃；更加可能是造成其他变量的不准确而引起其他的错误。所以在声明变量后马上对其 初始化一个默认的值是最简单有效的办法，否则项目大了你找都没地方找。代码存在错误在debug方式下可能会忽略而不被察觉到。debug方式下数组越界也大多不会出错，在release中就暴露出来了，这个找起来就比较难了。   
2. 自定义消息的消息参数。   
   MFC为我们提供了很好的消息机制，更增加了自定义消息，好处我就不用多说了。这也存在debug跟release的问 题吗？答案是肯定的。在自定义消息的函数体声明时，时常会看到这样的写法：afx\_msg LRESULT OnMessageOwn(); Debug 情况下一般不会有任何问题，而当你在Release下且多线程或进程间使用了消息传递时就会导致无效句柄之类的错误。导致这个错误直接原因是消息体的参数 没有添加，即应该写 成：afx\_msg LRESULT OnMessageOwn(WPARAM wparam, LPARAM lparam); 3. release模式下不出错，但debug模式下报错。   
    这种情况下大多也是因为代码书写不正确引起的，查看MFC 的源码，可以发现好多ASSERT的语句(断言)，这个宏只是在debug模式下才有效，那么就清楚了，release版不报错是忽略了错误而不是没有错误，这可能存在很大的隐患，因为是Debug模式下，比较方便调试，好好的检查自己的代码，再此就不多说了。   
3. ASSERT, VERIFY, TRACE.......... 调试宏   
    这种情况很容易解释。举个例子：请在VC下输入ASSERT然后选中按F12跳到宏定义的地方，这里你就能够发现Debug中 ASSERT要执行AfxAssertFailedLine，而Release下的宏定义却为\"#define ASSERT(f) ((void)0)\"。所以注意在这些调试宏的语句不要用程序相关变量如i++写操作的语句。   
VERIFY 是个例外，\"#define VERIFY(f) ((void)(f))\"，即执行。   
  
哪些情况下Release版会出错？    
1.  Runtime  Library：链接哪种运行时刻函数库通常只对程序的性能产生影响。调试版本 的  Runtime  Library  包含了调试信息，并采用了一些保护机制以帮助发现错误，因此性能不如发布版本。编译器提供 的  Runtime  Library  通常很稳定，不会造成  Release  版错误；倒是由 于  Debug  的  Runtime  Library  加强了对错误的检测，如堆内存分配，有时会出现  Debug  有错 但  Release  正常的现象。应当指出的是，如果  Debug  有错，即使  Release  正常，程序肯定是有  Bug  的，只不 过可能是  Release  版的某次运行没有表现出来而已。   
    
2.  优化：这是造成错误的主要原因，因为关闭优化时源程序基本上是直接翻译的，而打开优化后编译器会作出一系列假设。这类错误主要有以下几种：    
(1)  帧指针 (Frame  Pointer)省略（简称  FPO  ）：在函数调用过程中，所有调用信息（返回地址、参数）以及自动变量都是放在栈中的。若函数的声明与实现不同（参数、返回值、调用方式），就会产生错误，但  Debug  方式下，栈的访问通过  EBP  寄存器保存的地址实现，如果没 有发生数组越界之类的错误（或是越界“不多”），函数通常能正常执行；Release  方式下，优化会省略  EBP  栈基址指针，这样通过一个全局 指针访问栈就会造成返回地址错误是程序崩溃。C++  的强类型特性能检查出大多数这样的错误，但如果用了强制类型转换，就不行了。你可以 在  Release  版本中强制加入  /Oy-  编译选项来关掉帧指针省略，以确定是否此类错误。此类错误通常有：    
  
MFC消息响应函数书写错误。正确的应为    
afx\_msg  LRESULT  OnMessageOwn(WPARAM  wparam,  LPARAM  lparam);    
ON\_MESSAGE  宏 包含强制类型转换。防止这种错误的方法之一是重定义  ON\_MESSAGE  宏，把下列代码加到  stdafx.h  中 （在#include  "afxwin.h"之后）,函数原形错误时编译会报错

**C++代码  [收藏代码](javascript:void())**

1. #undef  ON\_MESSAGE
2. #define  ON\_MESSAGE(message,  memberFxn)  \
3. {  message,  0,  0,  0,  AfxSig\_lwl,  \
4. (AFX\_PMSG)(AFX\_PMSGW)(**static\_cast**<  **LRESULT**  (AFX\_MSG\_CALL  \
5. CWnd::\*)(**WPARAM**,  **LPARAM**)  >  (&memberFxn)  },

(2)  volatile  型变量：volatile  告诉编译器该变量可能被程序之外的未知方式修改（如系统、其他进程和线程）。优化程序为了使程序性能提高，常把一些变量放在寄存器中（类似于  register  关键字），而其他进程只能对该变量所在的内存进行修改，而寄存器中的 值没变。如果你的程序是多线程的，或者你发现某个变量的值与预期的不符而你确信已正确的设置了，则很可能遇到这样的问题。这种错误有时会表现为程序在最快 优化出错而最小优化正常。把你认为可疑的变量加上  volatile  试试。    
  
(3)  变量优化：优化程序会根据变量的使用情况优化变量。例如，函数中有一个未被使用的变量，在  Debug  版中它有可能掩盖一个数组越界，而在  Release  版中，这个变量很可能被 优化调，此时数组越界会破坏栈中有用的数据。当然，实际的情况会比这复杂得多。与此有关的错误有：    
    非法访问，包括数组越界、指针错误 等。例如

**C++代码  [收藏代码](javascript:void())**

1. **void**  fn(**void**)
2. {
3. **int**  i;
4. i  =  1;
5. **int**  a[4];
6. {
7. **int**  j;
8. j  =  1;
9. }
10. a[-1]  =  1; //当然错误不会这么明显，例如下标是变量
11. a[4]  =  1;
12. }

j  虽然在数组越界时已出了作用域，但其空间并未收回，因而  i  和  j  就会掩盖越界。而  Release  版由于  i、j  并未其很大作用可能会被优化掉，从而使栈被破坏。    
  
3.  \_DEBUG  与  NDEBUG  ：当定义了  \_DEBUG  时，assert()  函数会被编译， 而  NDEBUG  时不被编译。除此之外，VC++中还有一系列断言宏。这包括：    
  
ANSI  C  断 言  void  assert(int  expression  );    
C  Runtime  Lib  断 言  \_ASSERT(  booleanExpression  );    
\_ASSERTE(  booleanExpression  );    
MFC  断 言  ASSERT(  booleanExpression  );    
VERIFY(  booleanExpression  );    
ASSERT\_VALID(  pObject  );    
ASSERT\_KINDOF(  classname,  pobject  );    
ATL  断 言  ATLASSERT(  booleanExpression  );    
此外，TRACE()  宏的编译也受  \_DEBUG  控 制。    
  
所有这些断言都只在  Debug版中才被编译，而在  Release  版中被忽略。唯一的例外 是  VERIFY()  。事实上，这些宏都是调用了  assert()  函数，只不过附加了一些与库有关的调试代码。如果你在这些宏中加入了任何 程序代码，而不只是布尔表达式（例如赋值、能改变变量值的函数调用  等），那么  Release  版都不会执行这些操作，从而造成错误。初学者很容 易犯这类错误，查找的方法也很简单，因为这些宏都已在上面列出，只要利用  VC++  的  Find  in  Files  功能在工程所有文件中 找到用这些宏的地方再一一检查即可。另外，有些高手可能还会加入  #ifdef  \_DEBUG  之类的条件编译，也要注意一下。    
顺便值 得一提的是  VERIFY()  宏，这个宏允许你将程序代码放在布尔表达式里。这个宏通常用来检查  Windows  API  的返回值。有些人 可能为这个原因而滥用  VERIFY()  ，事实上这是危险的，因为  VERIFY()  违反了断言的思想，不能使程序代码和调试代码完全分离， 最终可能会带来很多麻烦。因此，专家们建议尽量少用这个宏。    
  
一、"Debug是调试版本，包括的程序信息更多"    
补充：只有DEBUG版的程序才能设置断点、单步执行、使用 TRACE/ASSERT等调试输出语句。REALEASE不包含任何调试信息，所以体积小、运行速度快。   
  
I.    内存分配问题     
   1.    变量未初始化。下面的程序在debug中运行的很好。

**C++代码  [收藏代码](javascript:void())**

1. thing    \*    search(thing    \*    something)
2. **BOOL**    found;
3. **for**(**int**    i    =    0;    i    <    whatever.GetSize();    i++)
4. {
5. **if**(whatever[i]->field    ==    something->field)
6. {    /\*    found    it    \*/
7. found    =    TRUE;
8. **break**;
9. }    /\*    found    it    \*/
10. }
11. **if**(found)
12. **return**    whatever[i];
13. **else**
14. **return**    NULL;

   而在release中却不行，因为debug中会自动给变量初始化found=FALSE,而在release版中则不会。所以尽可能的给变量、类或结构 初始化。     
      
   2.    数据溢出的问题     
   如：

**C++代码  [收藏代码](javascript:void())**

1. **char**    buffer[10];
2. **int**    counter;
3. lstrcpy(buffer,    "abcdefghik");

   在debug版中buffer的NULL覆盖了counter的高位，但是除非counter>16M,什么问题也 没有。但是在release版 中，counter可能被放在寄存器中，这样NULL就覆盖了buffer下面的空间，可能就是函数的返回地址，这将导致 ACCESS    ERROR。     
      
   3.    DEBUG版和RELEASE版的内存分配方式是不同的    。如果你在DEBUG版中申请    ele    为    6\*sizeof(DWORD)=24bytes,实际上分配给你的是32bytes（debug版以32bytes为单位分配），    而在release版，分配给你的就是24bytes（release版以8bytes为单位），所以在debug版中如果你写ele[6],可能不会有 什么问题，而在release版中，就有ACCESS    VIOLATE。     
      
   II.    ASSERT和VERIFY     
      
   1.    ASSERT在Release版本中是不会被编译的。     
      
   ASSERT宏是这样定义的

**C++代码  [收藏代码](javascript:void())**

2. #ifdef    \_DEBUG
3. #define    ASSERT(x)    if(    (x)    ==    0)    report\_assert\_failure()
4. #else
5. #define    ASSERT(x)
6. #endif

   实际上复杂一些，但无关紧要。假如你在这些语句中加了程序中必须要有的代码     
   比如

**C++代码  [收藏代码](javascript:void())**

2. ASSERT(pNewObj    =    **new**    CMyClass);
4. pNewObj->MyFunction();

   这种时候Release版本中的pNewObj不会分配到空间     
      
   所以执行到下一个语句的时候程序会报该程序执行了非法操作的错误。这时可以用VERIFY    ：

**C++代码  [收藏代码](javascript:void())**

2. #ifdef    \_DEBUG
3. #define    VERIFY(x)    if(    (x)    ==    0)    report\_assert\_failure()
4. #else
5. #define    VERIFY(x)    (x)
6. #endif

   这样的话，代码在release版中就可以执行了。     
      
   III.    参数问题：     
      
   自定义消息的处理函数，必须定义如下：     
   afx\_msg    LRESULT    OnMyMessage(WPARAM,    LPARAM);     
   返回值必须是HRESULT型，否则Debug会过，而Release出错     
      
   IV.    内存分配     
   保证数据创建和清除的统一性：如果一个DLL提供一个能够创建数据的函数，那么这个DLL同时应该提供一个函数销毁这些数据。数据的创建和清除应该在同一 个层次上。     
      
   V.    DLL的灾难     
   人们将不同版本DLL混合造成的不一致性形象的称为    “动态连接库的地狱“(DLL    Hell)    ，甚至微软自己也这么说 <http://msdn.microsoft.com/library/techart/dlldanger1.htm)>。     
      
   如果你的程序使用你自己的DLL时请注意：     
      
   1.    不能将debug和release版的DLL混合在一起使用。debug都是debug版，release版都是release版。     
   解决办法是将debug和release的程序分别放在主程序的debug和release目录下     
      
   2.    千万不要以为静态连接库会解决问题，那只会使情况更糟糕。     
      
   VI.    RELEASE版中的调试    ：     
   1.    将ASSERT()    改为    VERIFY()    。找出定义在"#ifdef    \_DEBUG"中的代码，如果在RELEASE版本中需要这些代码请将他们移到定义外。查找TRACE(...)中代码，因为这些代码在RELEASE中 也不被编译。    请认真检查那些在RELEASE中需要的代码是否并没有被便宜。     
   2.    变量的初始化所带来的不同，在不同的系统，或是在DEBUG/RELEASE版本间都存在这样的差异，所以请对变量进行初始化。     
   3.    是否在编译时已经有了警告?请将警告级别设置为3或4,然后保证在编译时没有警告出现.     
  
   VII.    将Project    Settings"    中    "C++/C    "    项目下优化选项改为Disbale（Debug）。编译器的优化可能导致许多意想不到的错误，请参http://www.pgh.net/~newcomer/debug\_release.htm     
      
   1.    此外对RELEASE版本的软件也可以进行调试，请做如下改动：     
      
   在"Project Settings"中"C++/C"项目下设置"category"为"General"并且将"Debug    Info"设置为"Program Database"。     
      
   在"Link"项目下选中"Generate Debug Info"检查框。     
      
   "Rebuild All"     
      
   如此做法会产生的一些限制：     
      
   无法获得在MFC DLL中的变量的值。     
      
   必须对该软件所使用的所有DLL工程都进行改动。     
      
   另：     
      
   1. MS BUG：MS的一份技术文档中表明，在VC5中对于DLL的"Maximize    Speed"优化选项 并未被完全支持，因此这将会引起内存错误并导致程序崩溃。     
      
   2. http://www.sysinternals.com/有 一个程序DebugView，用来捕捉OutputDebugString的输出，运行起来后（估计是自设为system debugger）就可以观看所有程序的OutputDebugString的输出。此后，你可以脱离VC来运行你的程序并观看调试信息。     
      
   3.    有一个叫Gimpel    Lint的静态代码检查工具，据说比较好用http://www.gimpel.com/    不过要化$的。   
  
Debug与Release不同的问题在刚开始编写代码时会经常发生，99%是因为你的代码书写错误而导致的，所以不要动不动就说系统问 题或编译器问题，努力找找自己的原因才是根本。我从前就常常遇到这情况，经历过一次次的教训后我就开始注意了，现在我所写过的代码我已经好久没遇到这种问 题了。下面是几个避免的方面，即使没有这种问题也应注意一下：   
1. 注意变量的初始化，尤其是指针变量，数组变量的初始化(很大的情况下另作考 虑了)。   
2. 自定义消息及其他声明的标准写法   
3. 使用调试宏时使用后最好注释掉   
4. 尽量使用 try - catch(...)   
5. 尽量使用模块，不但表达清楚而且方便调试。