C++ debug 和 release 版本的区别及调试技巧

2018 年 06 月 21 日 19:08:30 心情第一 阅读数：802

一、Debug 和 Release 编译方式的本质区别

Debug 通常称为调试版本，它包含调试信息，并且不作任何优化，便于程序员调试程序。Release 称为发布版本，它往往是进行了各种优化，使得程序在代码大小和运行速度上都是最优的，以便用户很好地使用。

Debug 和 Release 的真正秘密，在于一组编译选项。下面列出了分别针对二者的选项（当然除此之外还有其他一些，如/Fd /Fo，但区别并不重要，通常他们也不会引起 Release 版错误，在此不讨论）

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(编辑继续)数据库，这样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

实际上，Debug 和 Release 并没有本质的界限，他们只是一组编译选项的集合，编译器只是按照预定的选项行动。事实上，我们甚至可以修改这些选项，从而得到优化过的调试版本或是带跟踪语句的发布版本。

二、哪些情况下 Release 版会出错

有了上面的介绍，我们再来逐个对照这些选项看看 Release 版错误是怎样产生的

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

```
#undef ON_MESSAGE
#define ON_MESSAGE(message, memberFxn)      { message, 0, 0, 0, AfxSig_lwl,
 (AFX_PMSG)(AFX_PMSGW)(static_cast< LRESULT (AFX_MSG_CALL      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

```
void fn(void)
{
  int i;
  i = 1;
  int a[4];
  {
    int j;
```

```
        j = 1;
      }
      a[-1] = 1;//当然错误不会这么明显，例如下标是变量
      a[4] = 1;
    }
```

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。

4. /GZ 选项：这个选项会做以下这些事

(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。

除此之外，/Gm /GF 等选项造成错误的情况比较少，而且他们的效果显而易见，比较容易发现。

三、怎样“调试” Release 版的程序

遇到 Debug 成功但 Release 失败，显然是一件很沮丧的事，而且往往无从下手。如果你看了以上的分析，结合错误的具体表现，很快找出了错误，固然很好。但如果一时找不出，以下给出了一些在这种情况下的策略。

1. 前面已经提过，Debug 和 Release 只是一组编译选项的差别，实际上并没有什么定义能区分二者。我们可以修改 Release 版的编译选项来缩小错误范围。如上所述，可以把 Release 的选项逐个改为与之相对的 Debug 选项，如 /MD 改为 /MDd、/O1 改为 /Od，或运行时间优化改为程序大小优化。注意，一次只改一个选项，看改哪个选项时错误消失，再对应该选项相关的错误，针对性地查找。这些选项在 Project\Settings... 中都可以直接通过列表选取，通常不要手动修改。由于以上的分析已相当全面，这个方法是最有效的。

2. 在编程过程中就要时常注意测试 Release 版本，以免最后代码太多，时间又很紧。

3. 在 Debug 版中使用 /W4 警告级别，这样可以从编译器获得最大限度的错误信息，比如 if(i =0)就会引起 /W4 警告。不要忽略这些警告，通常这是你程序中的 Bug 引起的。但有时 /W4 会带来很多冗余信息，如 未使用的函数参数 警告，而很多消息处理函数都会忽略某些参数。我们可以用

```
#progma warning(disable: 4702) //禁止
//...
#progma warning(default: 4702) //重新允许
```

来暂时禁止某个警告，或使用

```
#progma warning(push, 3) //设置警告级别为 /W3
//...
#progma warning(pop) //重设为 /W4
```

来暂时改变警告级别，有时你可以只在认为可疑的那一部分代码使用 /W4。

4.你也可以像 Debug 一样调试你的 Release 版，只要加入调试符号。在 Project/Settings... 中，选中 Settings for "Win32 Release"，选中 C/C++ 标签，Category 选 General，Debug Info 选 Program Database。再在 Link 标签 Project options 最后加上 "/OPT:REF" (引号不要输)。这样调试器就能使用 pdb 文件中的调试符号。但调试时你会发现断点很难设置，变量也很难找到——这些都被优化过了。不

过令人庆幸的是，Call Stack 窗口仍然工作正常，即使帧指针被优化，栈信息（特别是返回地址）仍然能找到。这对定位错误很有帮助。