IAR 的 Workspace 顶部下拉菜单中 Debug 和 Release 区别

Hanson-he

在 IAR 的 Workspace 窗口顶部的下拉菜单中有两个选项，Debug 和 Release。名字和数量可以在菜单栏的 Project-->Edit Configuration 中增删修改。每个选项都对应着一种配置（也就是项目名称右击 Options 里的内容），互相是独立的。所以用起来很方便，直接在不同的配置间切换。

realse 版本与 debug 版本的区别如下：

DEBUG 版本和 RELEASE 版本的区别：DEBUG 版本里会包括基本的调试信息，供开发和调试使用；而 RELEASE 版本不会包括调试信息，一般是发布给用户使用。所以你会发现 DEBUG 版本的 SIZE 比 RELEASE 版本会大许多。另外，RELEASE 版本在编译链接的时候检查会更严格，所以有时候会出现 DEBUG 版本能使用，而 RELEASE 版本却会出现一些问题的情况。编译器优化级别，链接器输出文件的格式(是否包含调试信息等) ，当然，debug 和 release 这两种 build configuration 的属性，在 project 创建完之后是可以任意修改的。

虽然可能在 Release 版本下没有问题，但是并不等于说程序没有 Bug，只是由于运行库进行了优化和丢弃，问题没有明显的展现出来，这样隐含式的错误后期很难再找出来，即使发现了，也由于编码的原因，修改困难。所以考虑还是在开发的阶段尽量的用 debug 方式编译和修改，当在 debug 模式下没有问题以后再改为 release 生成给客户运营用。

一、Debug 和 Release 编译方式的本质差别

Debug 通常称为调试版本，他包含调试信息，并且不作所有优化，便于程式员调试程式。Release 称为发布版本，他往往是进行了各种优化，使得程式在代码大小和运行速度上都是最优的，以便用户非常好地使用。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（当然除此之外更有其他一些，如/Fd /Fo，但差别并不重要，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原始码不需重新编译;/GZ 能帮助捕捉内存错误;/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，使程式最小或最快,/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，我们甚至能修改这些选项，从而得到优化过的调试版本或是带跟踪语句的发布版本。

再来逐个对照这些选项看看 Release 版错误是怎样产生。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。

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：

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) }。

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。

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掉，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。

_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。

/gz 选项的作用如下：

(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。

IAR 编译 Release 方法:通常写程序，调试程序都是在 Debug 下。当程序写完后，切换到 Release 下编译，很多时候，会有一大堆错误。原因 Debug 与 release 设置不一样。在 Debug 下写程序的过程中，对 debug 的设置有改动，而 relase 的设置没有变。一个方法可以把 debug 的设置复制到 relase 上。先把 Release 删除掉，然后以 Debug 为模板，新建一个 Release。