[C语言之断言](http://blog.csdn.net/wxq1987525/article/details/6639220)

标签： [语言](http://www.csdn.net/tag/%e8%af%ad%e8%a8%80)[c](http://www.csdn.net/tag/c)[report](http://www.csdn.net/tag/report)[function](http://www.csdn.net/tag/function)[file](http://www.csdn.net/tag/file)[string](http://www.csdn.net/tag/string)

2011-07-27 21:38 9698人阅读 [评论](http://blog.csdn.net/wxq1987525/article/details/6639220#comments)(7) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/wxq1987525/article/details/6639220#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

c++（11） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

[**C语言的那些小秘密之断言**](http://blog.csdn.net/bigloomy/article/details/6614491)

每次写摘要我都觉得是一件很头疼的事儿，因为我知道摘要真的很重要，它几乎直接就决定了读者的数量。可能花了九六二虎之力写出来的东西，因为摘要的失败而前功尽弃，因为绝大多数的读者看文章之前都会浏览下摘要，如果他们发现摘要“不对口”，没有什么特色和吸引人的地方，那么轻则采用一目十行的方法看完全文，重则对文章判“死刑”，一篇文章的好坏虽然不能用摘要来衡量，但是它却常常被读者用来衡量一篇文章的好坏，从而成为了文章读者数量多少的一个关键因素。下面言归正传来说说断言，如果出于一般性的学习C语言，应付考试的话，我想很少有人会在代码中使用断言，可能有的人在此之前从来没有使用过断言。那么断言的使用到底能给我们的代码带来什么呢？我尽可能的把我所理解的断言的使用讲解清楚，希望我在此所讲的断言能够对你有所帮助，让你以后能够在代码中灵活使用断言。

在讲解之前，我们先来对断言做一个基本的介绍，让大家对断言有一个大致的了解。在使用C语言编写工程代码时，我们总会对某种假设条件进行检查，断言就是用于在代码中捕捉这些假设，可以将断言看作是异常处理的一种高级形式。断言表示为一些布尔表达式，程序员相信在程序中的某个特定点该表达式值为真。可以在任何时候启用和禁用断言验证，因此可以在测试时启用断言，而在部署时禁用断言。同样，程序投入运行后，最终用户在遇到问题时可以重新起用断言。它可以快速发现并定位软件问题，同时对系统错误进行自动报警。断言可以对在系统中隐藏很深，用其它手段极难发现的问题可以用断言来进行定位，从而缩短软件问题定位时间，提高系统的可测性。实际应用时，可根据具体情况灵活地设计断言。

通过上面的讲解我们对于断言算是有了一个大概的了解，那么接下来我们就来看看C语言中assert宏在代码中的使用。

原型定义：

void assert( int expression );

assert宏的原型定义在<assert.h>中，其作用是先计算表达式 expression ，如果expression的值为假（即为0），那么它先向stderr打印一条出错信息，然后通过调用abort 来终止程序运行。

下面来看看一段代码：

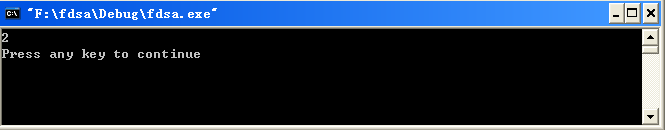
#include <stdio.h>  
#include <assert.h>

int main( void )  
{  
      int i;  
   i=1;  
   assert(i++);

   printf("%d\n",i);

       return 0;  
}

 运行结果为：



看看运行结果，因为我们给定的i初始值为1，所以使用assert(i++);语句的时候不会出现错误，进而执行了i++，所以其后的打印语句输出值为2。如果我们把i的初始值改为0，那么就回出现如下错误。  
Assertion failed: i++, file E:\fdsa\assert2.cpp, line 8  
Press any key to continue

是不是发现根据提示很快就能定位出错点呢？！既然assert这么便于定位出错点，看来的确我们有必要熟练的在代码中使用它，但是什么东西的使用都是有规则的，assert的使用也不例外。

断言语句不是永远会执行，可以屏蔽也可以启用，这就要求assert不管是在屏蔽还是启用的情况下都不能对我们本身代码的功能有所影响，这样的话刚才我们在代码中使用了一句assert(i++);是不妥的，因为我们一旦禁用了assert，i++的语句就得不到执行，对于接下来i值的使用就会出现问题了，所以对于这样的语句我们应该是要分开来实现，写出如下两句来替代， assert(i); i++;，所以这就对于断言的使用有了相应的要求，那么我们一般在什么情况下使用断言呢？主要体现在一下几个方面：

1.可以在预计正常情况下程序不会到达的地方放置断言。（如assert (0);）

2.使用断言测试方法执行的前置条件和后置条件 。

3.使用断言检查类的不变状态，确保任何情况下，某个变量的状态必须满足。（如某个变量的变化范围）

对于上面的前置条件和后置条件可能有的读者还不是很了解，那么看看下面的解释你就明白了。

前置条件断言：代码执行之前必须具备的特性

后置条件断言：代码执行之后必须具备的特性

前后不变断言：代码执行前后不能变化的特性

当然在使用的断言的过程中会有一些我们应该注意的事项和养成一些良好的习惯，如：

1.每个assert只检验一个条件，因为同时检验多个条件时，如果断言失败，我们就无法直观的判断是哪个条件失败

2.不能使用改变环境的语句，就像我们上面的代码改变了i变量，在实际编写代码的过程中是不能这样做的

3.assert和后面的语句应空一行，以形成逻辑和视觉上的一致感，也算是一种良好的编程习惯吧，让编写的代码有一种视觉上的美感

4.有的地方，assert不能代替条件过滤

5.放在函数参数的入口处检查传入参数的合法性

6.断言语句不可以有任何边界效应

上面那么多的文字，似乎很枯燥，但是没办法，我们不能急功近利，还是要先坚持看完文字描述部分，这样在下面我们分析代码的过程中就能很快知道为什么会出现那样的问题了，也能在自己编写代码的时候熟练的使用assert，给自己的代码调试带来极大的便利，尤其是你在用C语言做工程项目的时候，如果你能够在你的代码中合理的使用assert，能使你创建更稳定、质量更好且不易于出错的代码。当需要在一个值为FALSE时中断当前操作的话，可以使用断言。单元测试必须使用断言，除了类型检查和单元测试外，断言还提供了一种确定各种特性是否在程序中得到维护的极好的方法。但凡优秀的程序员都能够在自己代码中很好的使用assert，编写出高质量的代码来。

说了assert这么多的有点，当然也要说说它的缺点了。

使用assert的缺点是，频繁的调用会极大的影响程序的性能，增加额外的开销。所以在调试结束后，可以通过在包含#include 的语句之前插入 #define NDEBUG 来禁用assert调用。

接下面分析一下下面的一段代码：

#include <stdio.h>  
//#define NDEBUG   
#include <assert.h>

int copy\_string(char from[],char to[])  
{  
 int i=0;  
 while(to[i++]=from[i]);

 printf("%s\n",to);

 return 1;  
}

int main()  
{  
 char str[]="this is a string!";  
 char dec\_str[206];

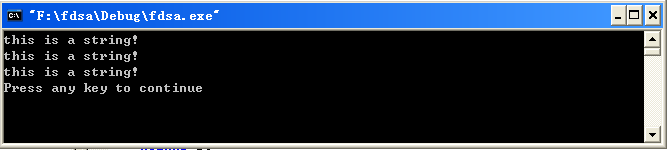
 printf("%s\n",str);

 assert(copy\_string(str,dec\_str));

 printf("%s\n",dec\_str);

 return 0;  
}

运行结果为：



在以上代码的开头部分我们把#define NDEBUG给注释掉了，所以我们启用了assert，main函数中使用了assert(copy\_string(str,dec\_str));来实现copy\_string函数的调用，在copy\_string函数中我们使用了一句return 1，所以最终的函数调用结果就等价于是assert(1)，所以接下来继续执行assert下面的打印语句，最终成功的打印了三条输出语句，如果我们把开头的注释部分打开，结果就只能成功的输出起始部分一条打印语句。

以上我们都是在围绕着assert宏在讲解，仅仅是教会大家如何来使用assert宏，那么接下来看看我们如何来实现自己的断言呢？

接下来我们看看另外一段代码：

#include <stdio.h>

//#undef  \_EXAM\_ASSERT\_TEST\_    //禁用  
#define  \_EXAM\_ASSERT\_TEST\_   //启用  
#ifdef \_EXAM\_ASSERT\_TEST\_     //启用断言测试  
 void assert\_report( const char \* file\_name, const char \* function\_name, unsigned int line\_no )  
{  
 printf( "\n[EXAM]Error Report file\_name: %s, function\_name: %s, line %u\n",   
         file\_name, function\_name, line\_no );

}  
 #define  ASSERT\_REPORT( condition )       \  
 do{       \  
 if ( condition )       \  
  NULL;        \  
 else         \  
  assert\_report( \_\_FILE\_\_, \_\_func\_\_, \_\_LINE\_\_ ); \  
 }while(0)  
 #else // 禁用断言测试   
#define ASSERT\_REPORT( condition )  NULL   
#endif /\* end of ASSERT \*/  
 int main( void )  
{  
    int i;  
    i=0;  
   // assert(i++);  
   ASSERT\_REPORT(i);  
     printf("%d\n",i);   
        return 0;  
}

运行结果如下：

[EXAM]Error Report file\_name: assert3.c, function\_name: main, line 29  
0  
细心的读者会发现我们并没有使用断言来结束当前程序的执行，所以在断言下面的printf成功的打印出了i的当前值，当然我们也可以做适当的修改，在断言出发现错误，那么就调用 abort();来使当前正在执行的程序异常终止，修改如下：

#include <stdio.h>  
#include <stdlib.h>

//#undef  \_EXAM\_ASSERT\_TEST\_    //禁用  
#define  \_EXAM\_ASSERT\_TEST\_   //启用  
#ifdef \_EXAM\_ASSERT\_TEST\_     //启用断言测试  
 void assert\_report( const char \* file\_name, const char \* function\_name, unsigned int line\_no )  
{  
 printf( "\n[EXAM]Error Report file\_name: %s, function\_name: %s, line %u\n",   
         file\_name, function\_name, line\_no );  
  abort();  
}

#define  ASSERT\_REPORT( condition )       \  
 do{       \  
 if ( condition )       \  
  NULL;        \  
 else         \  
  assert\_report( \_\_FILE\_\_, \_\_func\_\_, \_\_LINE\_\_ ); \  
 }while(0)

#else // 禁用断言测试   
#define ASSERT\_REPORT( condition )  NULL   
#endif /\* end of ASSERT \*/  
 int main( void )  
{  
    int i;  
    i=0;  
   // assert(i++);  
   ASSERT\_REPORT(i);  
    printf("%d\n",i);  
    return 0;

}

运行结果如下：

[EXAM]Error Report file\_name: assert3.c, function\_name: main, line 31  
Aborted  
此时就不会在执行接下来的打印语句了。看看我们自己的实现方式就知道，我们自己编写的断言可以比直接调用assert宏可以得到更多的信息量，主要是由于我们自己编写的断言更加的具有灵活性，可以根据自己的需要来打印输出不同的信息，同时也可以对于不同类型的错误或者警告信息使用不同的断言，这也是在工程代码中经常使用的做法。如果你在关注代码运行结果的同时也认真的阅读了我的代码，你会发现其中我在宏定义中使用了一个do{}while(0)，使用它有什么好处呢，或许在以上的代码中并没有体现出来，那么我们看看下面的代码你就知道了。

#include <stdio.h>

void print\_1(void)  
{  
 printf("print\_1\n");  
}  
void print\_2(void)  
{  
 printf("print\_2\n");  
}  
#define  printf\_value()    \  
   print\_1();   \  
   print\_2();   \

int main( void )  
{  
 int i=0;  
 if(i==1)  
 printf\_value();

 return 0;  
}

 运行结果：



还是备份一下文章描述，以防图片打开失败给读者带来困扰。

print\_2  
Press any key to continue

看了上面运行结果可能有的读者会很疑惑为什么会出现以上的错误呢？！if语句的条件不满足，那么print\_value()函数应该不会被调用啊，怎么会打印呢。如果我们把上面的printf\_value()替换为 print\_1();  print\_2();，就会很清楚的发现if语句在此的作用仅仅是不调用print\_1();，而print\_2();在控制之外，所以出现了上面的结果，有的读者可能会马上想到我们加上一个{}不就好了吗，在这里的确是加一个{}就可以了，因为这里是一个特殊情况，没有else语句，如果我们在以上的宏定义中使用{}，加入else语句后再来看看代码。

#include <stdio.h>

void print\_1(void)  
{  
 printf("print\_1\n");  
}

void print\_2(void)  
{  
 printf("print\_2\n");  
}

#define  printf\_value()    \  
  {     \  
  print\_1();   \  
  print\_2();}

int main( void )  
{  
 int i=0;  
 if(i==1)  
  printf\_value();  
 else  
  printf("add else word!!!");

 return 0;  
}

看似正确的代码，我们编译就会出现如下错误：

error C2181: illegal else without matching if

为什么会出现这样的错误呢？因为我们编写C语言代码时，在每个语句后面加分号是一种约定俗成的习惯，以上代码中我们在printf\_value()语句后面加了一个分号，正是由于这个分号的作用使得else没有与之相对应的if，所以编译出错。但是如果我们使用do{}while(0)就不会出现这些问题，所以我们在编写代码的时候应该学会在宏定义中使用do{}while(0)。

C语言断言内容的讲解到此就该结束了，上面内容已给出了在C语言编写代码的过程中断言较为详细的使用，其中后面使用我们自己实现的断言算得上是一个比较经典的断言设计方法了，读者可以在自己以后编写C语言代码的过程中参考下。由于本人水平有限，博客中的不妥或错误之处在所难免，殷切希望读者批评指正。同时也欢迎读者共同探讨相关的内容，如果乐意交流的话请留下你宝贵的意见