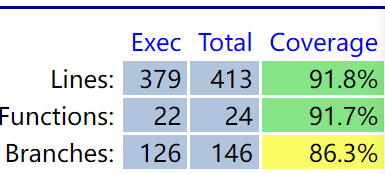
**白盒测试测试报告**

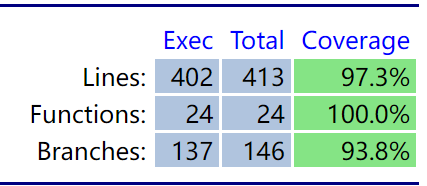
一、整体说明

本次白盒测试一共设计了7个测试用例，分别为：input1.txt，input2-1.txt，input2-2.txt，input3.txt，input4.txt，input5.txt，input6.txt

采用了gcov工具，测试用例中input1.txt的内容最多，最终得到的三种覆盖率详细结果如图：



为提高覆盖率，补充了剩余的其他的测试用例，最终得到的三种覆盖率详细结果如图：



二、结果分析

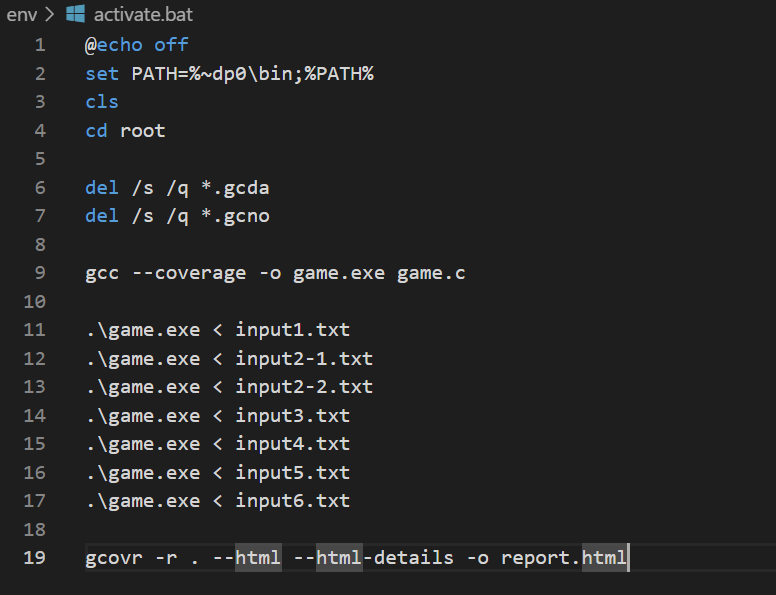
1.工具使用说明

gcov伴随gcc 发布。gcc编译加入-fprofile-arcs -ftest-coverage 参数生成二进制程序，执行测试用例生成代码覆盖率信息。

gcno文件：由-ftest-coverage产生的，包含了重建基本块图和相应的块的源码的行号的信息。

gcda文件：由加了-fprofile-arcs编译参数的编译后的文件运行所产生的，它包含了弧跳变的次数和其他的概要信息。

activate.bat：



@echo off: 批处理文件的开头，关闭命令行窗口中的命令回显，使输出更加清晰。

set PATH=%~dp0\bin;%PATH%: 这一行将批处理文件所在的目录下的 bin 子目录添加到系统的 PATH 环境变量中,确保编译和执行 game.exe 时可以找到所需的依赖项。

使用del /s /q \*.gcda和del /s /q \*.gcno命令删除当前目录及其子目录中的所有 .gcda 和 .gcno 文件。这两个文件会保留之前测试的覆盖率结果并把新的测试用例的结果累加进来，所以在进行一组新的测试时，先删除这两个文件。

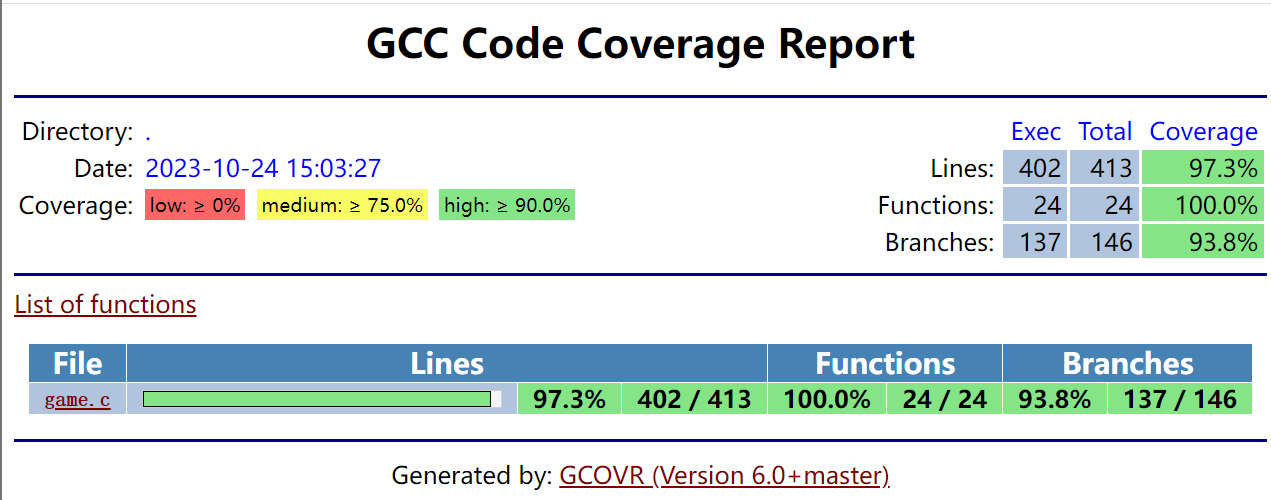
可以看到在gcc命令行中，使用的是--coverage参数，而不是-ftest-coverage和-fprofile-arcs参数，这是因为：--coverage等同于编译参数-fprofile-arcs -ftest-coverage以及在链接时增加-lgcov（-lgcov 选项，它用于链接 libgcov 库，这个库包含了生成报告所需的函数和数据）。

11-17行把7个测试用例作为输入进行测试。

最后使用gcovr工具生成代码覆盖率报告，’-r .’指定报告生成目录为当前目录，’-html’ 和 ‘--html-details’ 选项指示生成 HTML 格式的报告，并包括详细信息。 ‘-o report.html’ 选项指定了生成的报告文件的名称为 ‘report.html’。

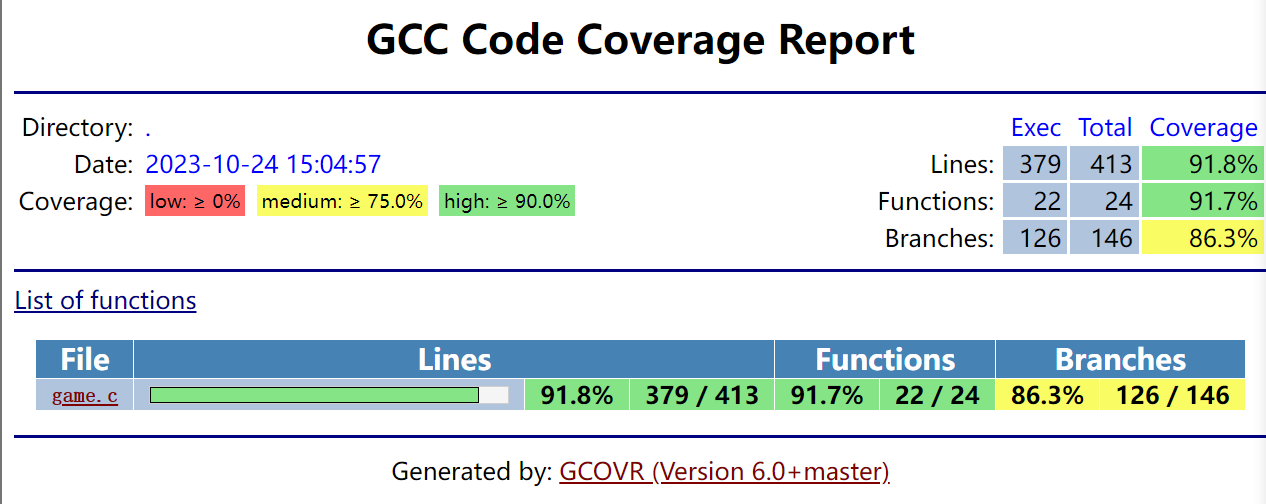
1. 对覆盖率报告的解释
2. 整体报告各项指标的含义

下图所示为整体测试覆盖率结果：



可以看到，共有三种覆盖率结果，分别为：Lines，Functions，Branches。其中语句覆盖率（Lines Coverage）达到了97.3%，函数覆盖率（Functions Coverage）达到了100%，分支覆盖率（Branches Coverage）达到了93.8%，三者均为高覆盖率结果（high：>=90%）。

下图为input1.txt单个测试用例的覆盖率结果：



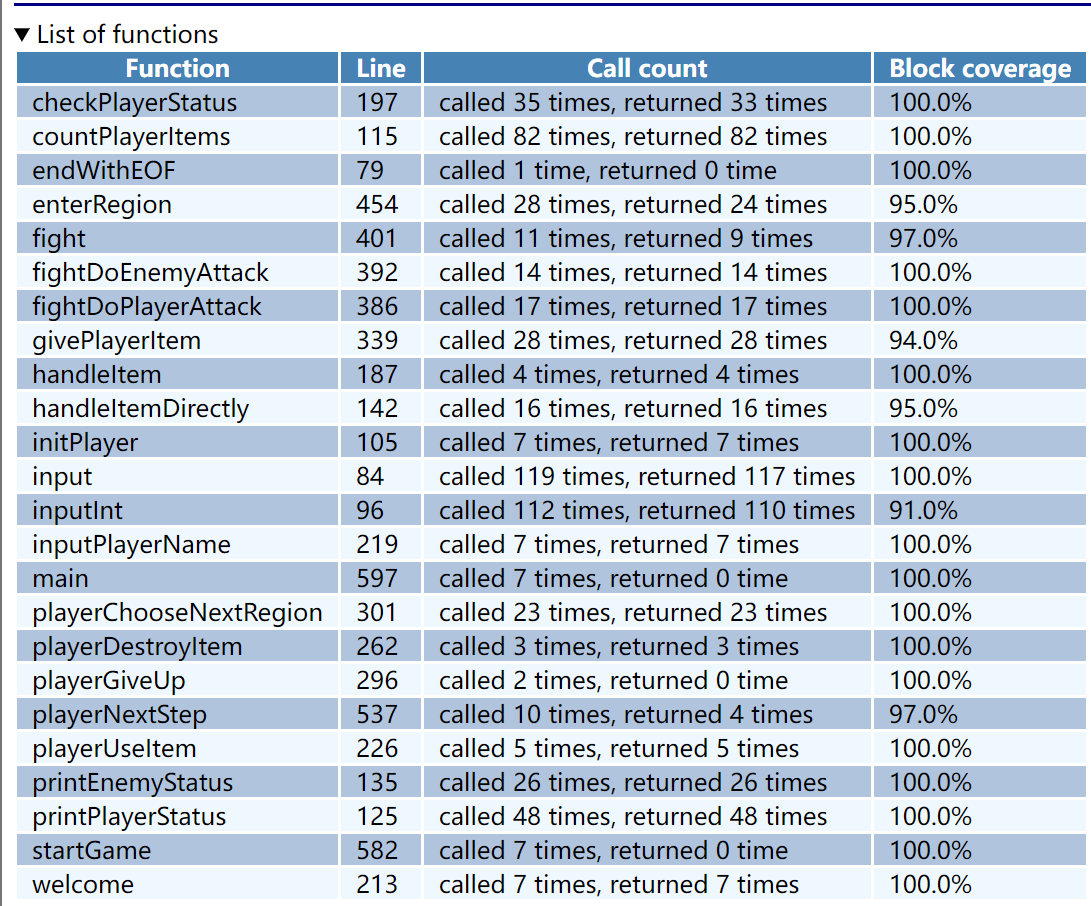
可以看到，input1.txt测试用例已经得到了很高的覆盖率，语句覆盖率达到91.8%，函数覆盖率达到91.7%，分支覆盖率达到86.3%。因此，在input1.txt的基础上，增加的测试用例均为内容较少的简单的测试用例，目的是补充input1.txt中无法涉及到的语句、函数和分支。

总体而言，所设计的这组测试用例得到的覆盖率结果相对较好。

1. 详细报告中各项指标的含义：

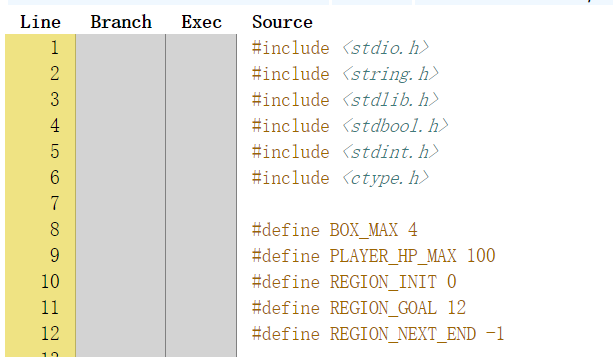
此处（(2)中）所展示的结果均为整组测试用例的结果。

首先，下图为game.c文件中函数的相关内容：



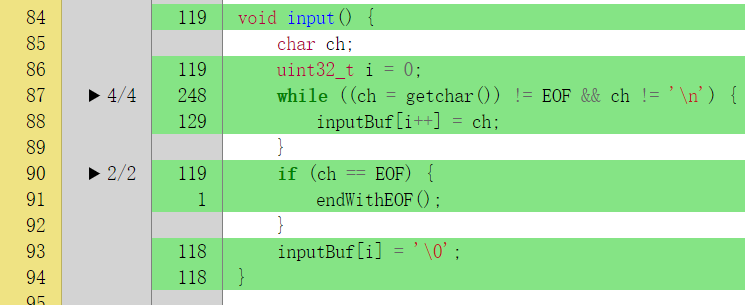
图中共有四列，Functions：函数名；Line：game.c文件中函数所在的行数；Call count：函数被调用的次数以及函数return的次数；

随后，在html-details报告中，如下图所示：



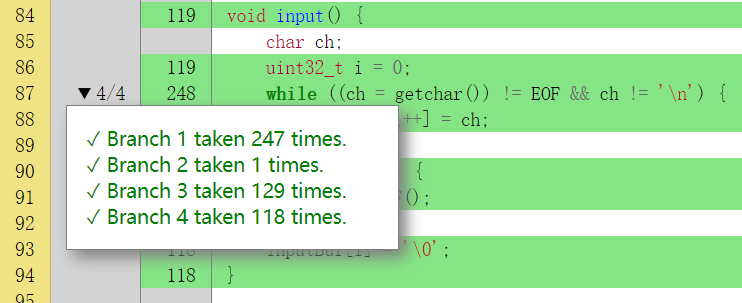
Line：表示在C文件中的行数；Branch：分支的执行情况；Exec：被执行的次数；Source：具体的代码。

以void input()函数为例：



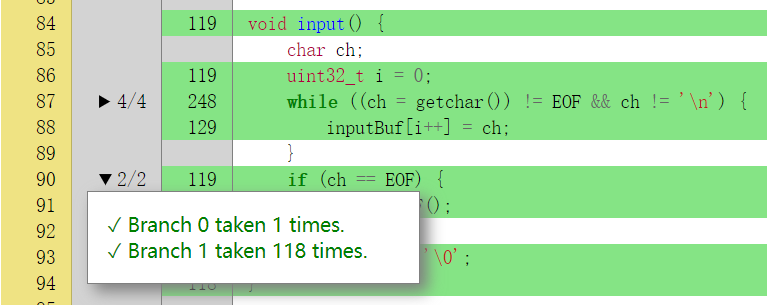
其中input()函数一共执行了119次，在函数内部，while语句的分支一共有4个，在测试用例中均被覆盖，if语句的分支共2个，同样在测试用例中被覆盖。

while语句4个分支的执行情况如下图所示：



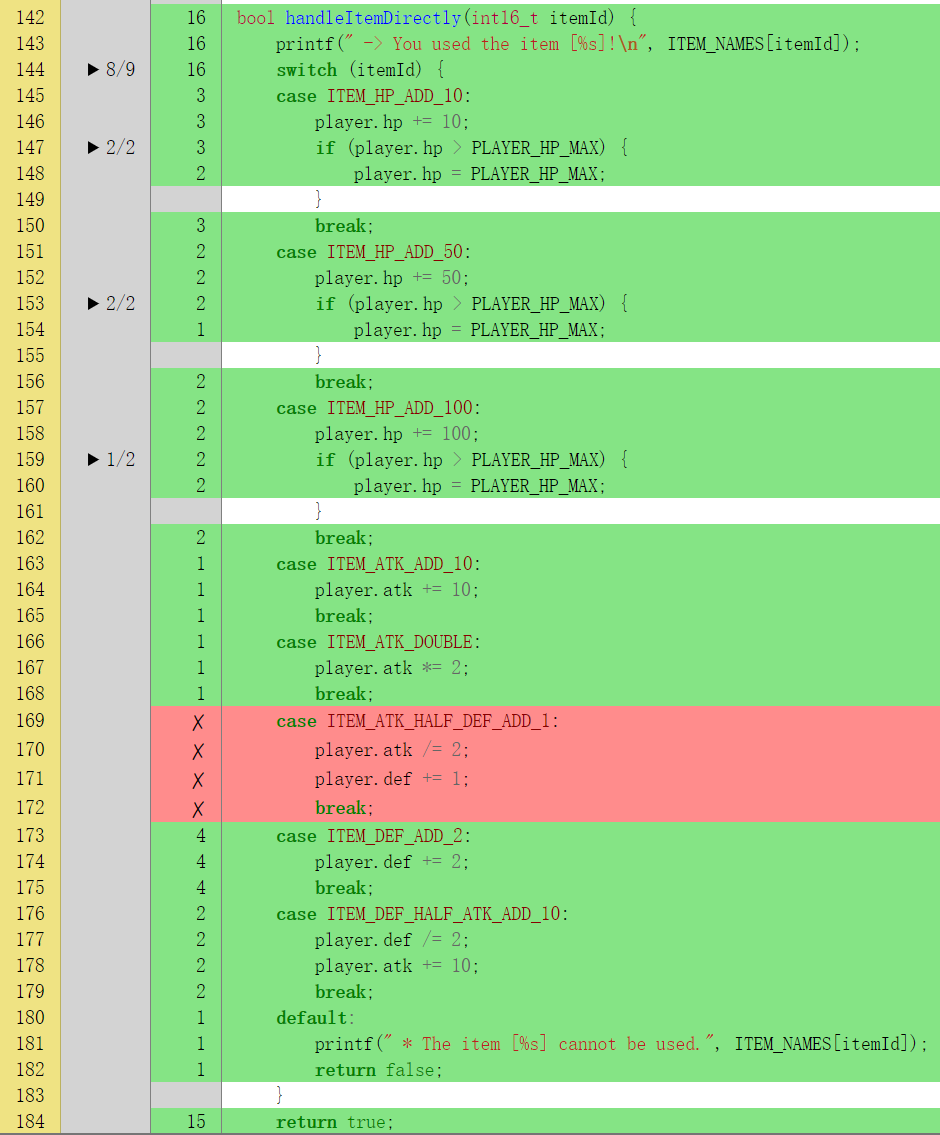
图中标出了分支的具体执行情况，根据本组测试用例，可以推断出，Branch1由Branch3和4组成，Branch1为条件判断为真，执行while循环内的语句的分支，Branch2为判断条件为假的分支（具体为ch == EOF的一次情况，即input5.txt的测试）。

if语句2个分支的具体执行情况如下图所示：



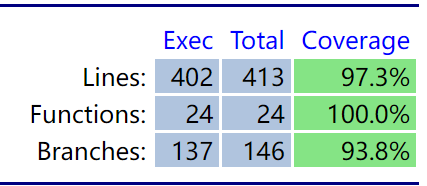
Branch0为ch == EOF的分支，Branch1为ch != EOF的分支。

下图所示为代码出现绿色和红色两种标色的情况，其中绿色表示该语句在整组测试用例中被执行过（在Exec列中也可以看到具体被执行的次数），红色表示该语句在整组测试用例中未被执行过（同样，在Exec列中标记为X）：



1. 未覆盖

整个测试得到的覆盖率结果如图：

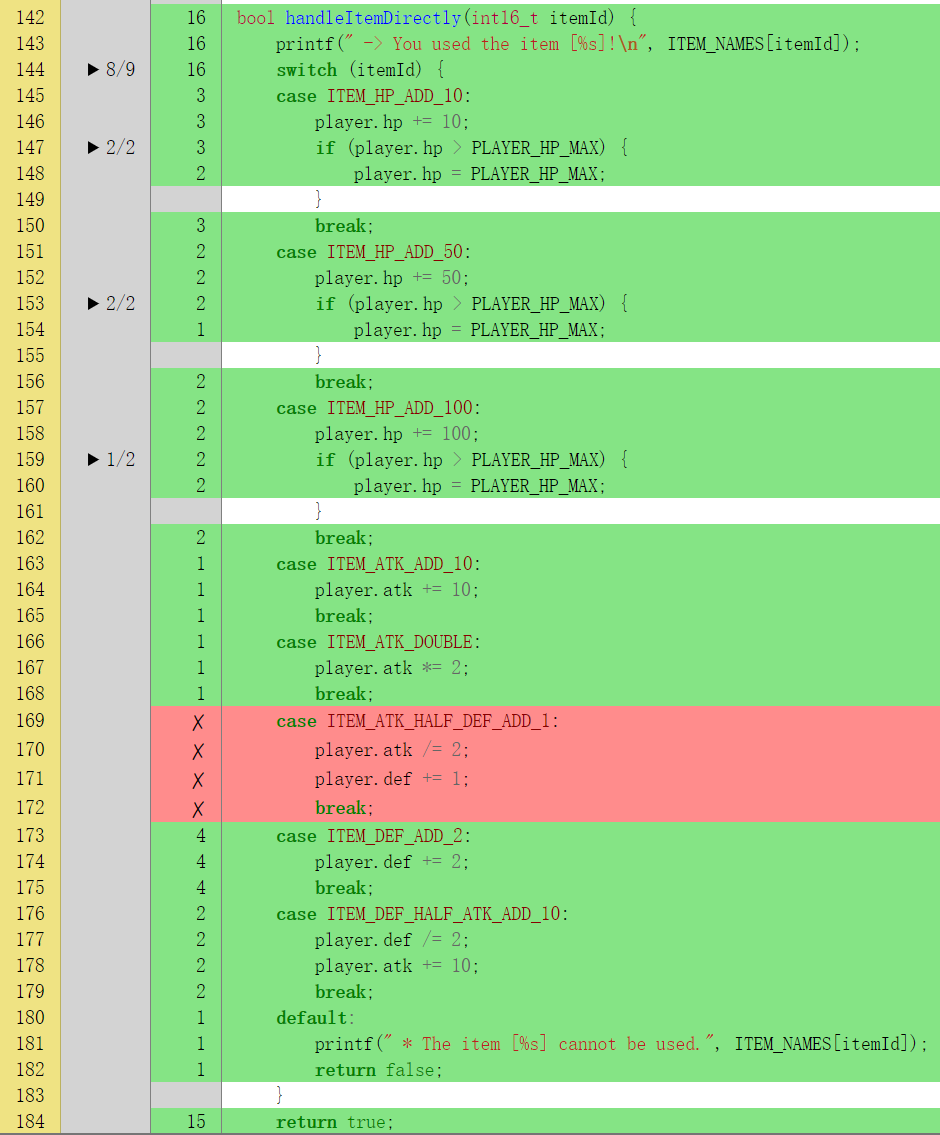


其中语句覆盖和分支覆盖中均有未覆盖到的情况，下面具体分析。

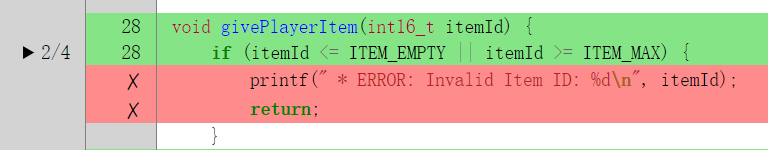
1. 语句

通过报告找到了未覆盖到的语句。

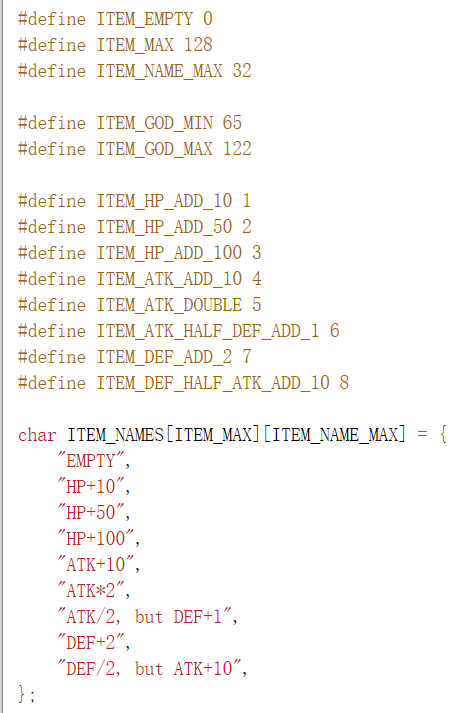
①在handleItemDirectly函数中的case ITEM\_ATK\_HALF\_DEF\_ADD\_1，道具"ATK/2, but DEF+1"没有被使用过。通过检查代码发现，该道具只存在于ITEM列表中，玩家得到道具的途径共两种（不包括天选之人），分别为进入宝箱区域，击败怪物。但是在这两种途径中，均无获得该道具的代码。



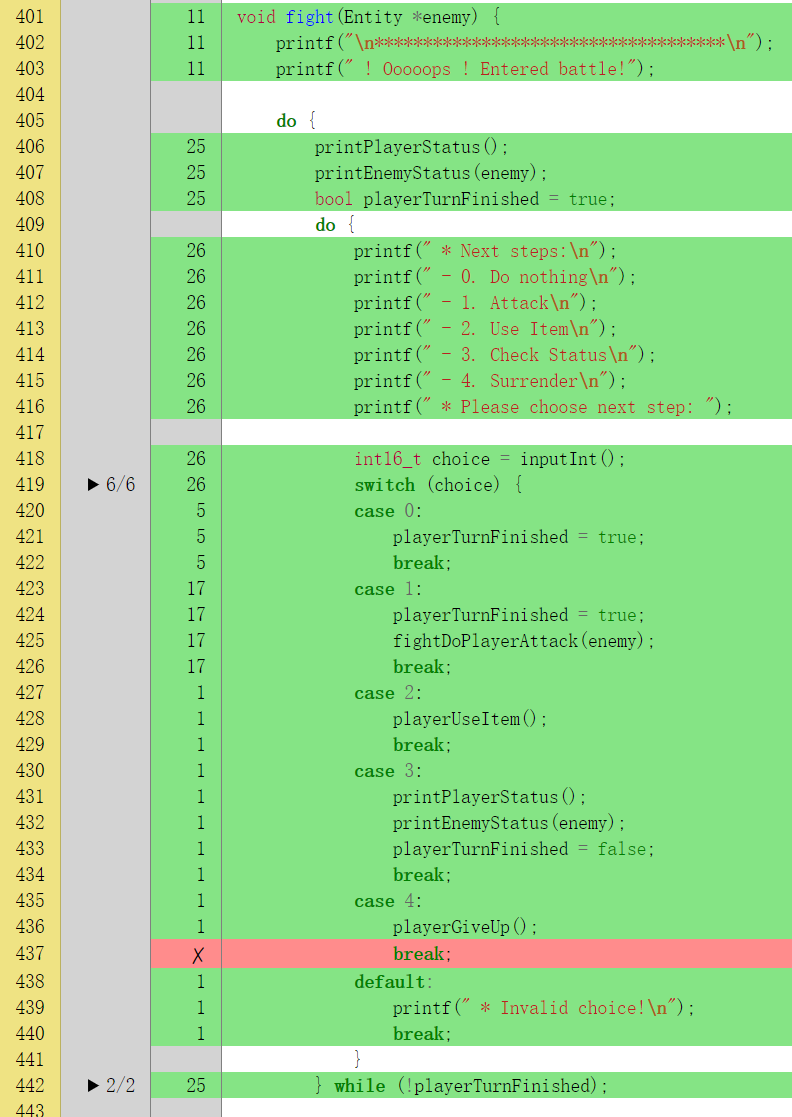
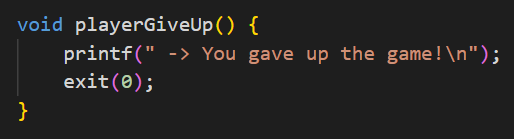
②：在givePlayerItem函数中，没有出现过Invalid Item ID的错误情况。



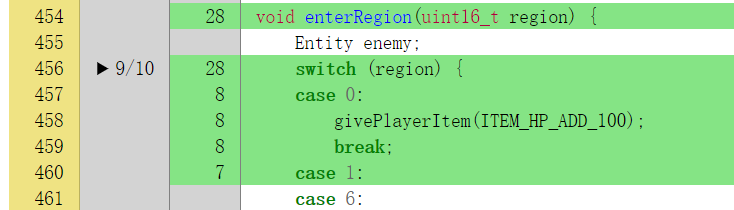
具体分析代码可以得知，所有可以得到的itemID如图所示，不存在itemID <= ITEM\_EMPTY或 itemID >= ITEM\_MAX的情况：

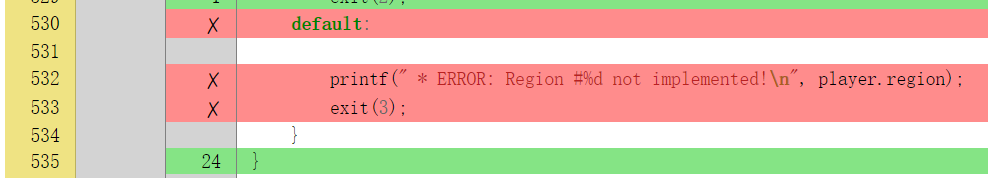


③：fight的give up情况，在player选择在对战中giveup，直接跳转到playerGiveUp()函数，该函数会直接exit(0)，所以break语句不会被执行：

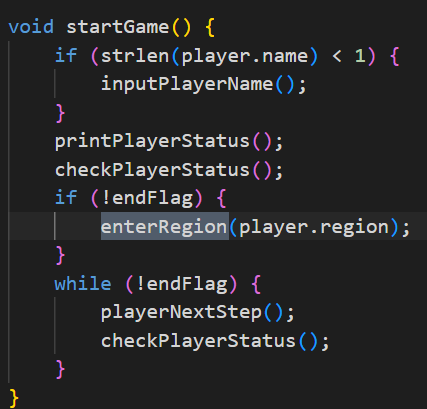


④：enterRegion函数中，switch case语句中的default情况不会出现。



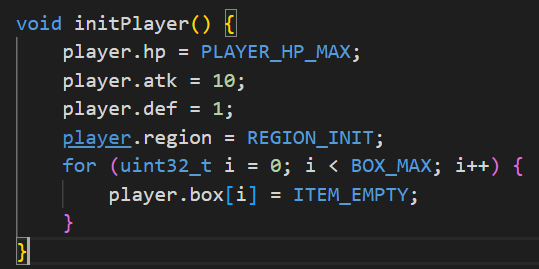


调用该函数的位置如下面两张图所示：

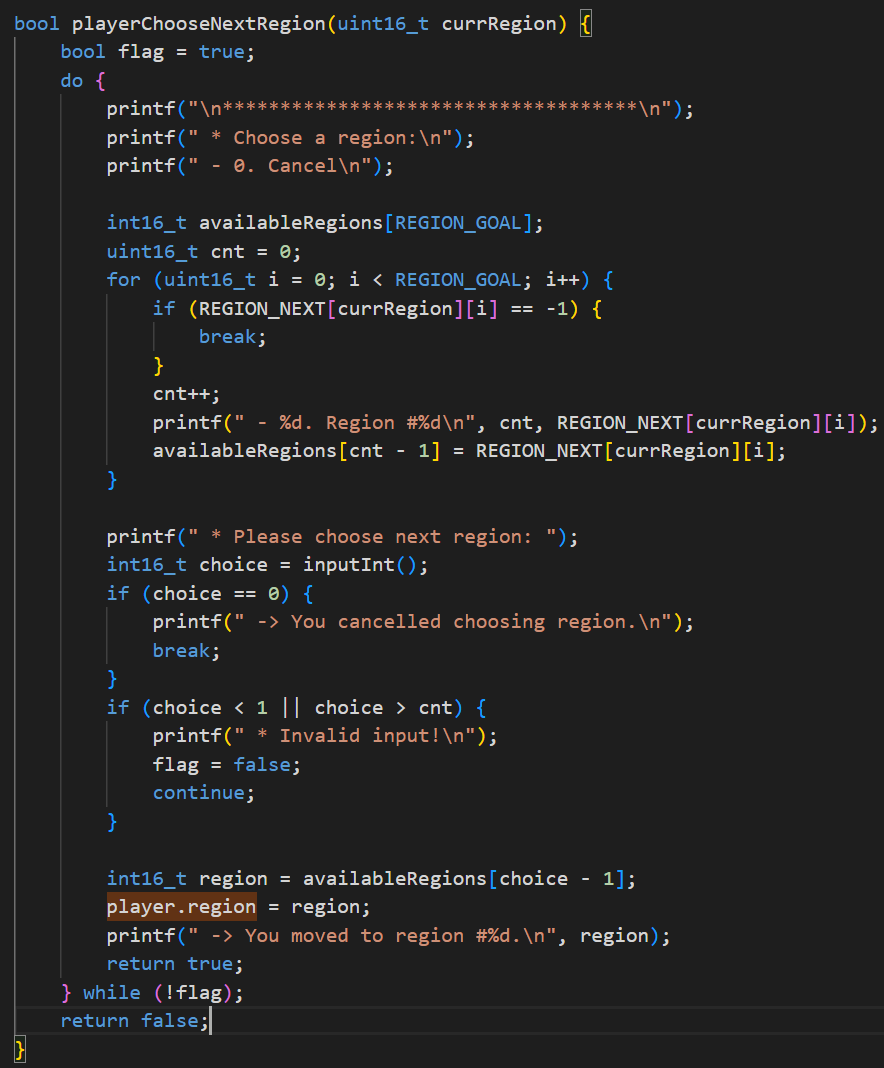




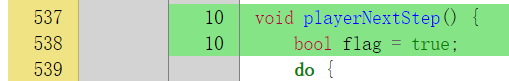
其中在startGame中的player.region为初始化的region，#define REGION\_INIT 0：



在playerChooseNextRegion中，choice在<1或>cnt的情况下会报Invalid input，并要求用户重新输入，cnt在for循环中进行++操作，而#define REGION\_GOAL 12，可以得到cnt最大为12，所以player.region不会出现enterRegion函数中default的情况。



⑤：在playerNextStep函数中，同③情况类似，playerGiveUp函数调用后直接exit(0)，break语句不会被执行。

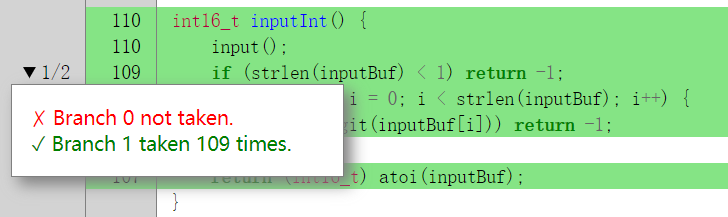




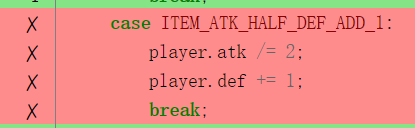
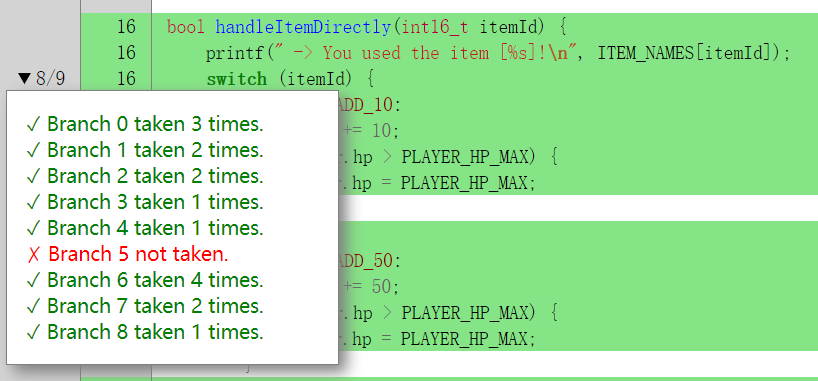
1. 分支

此处以函数为分序号的标准，并没有按照每个未覆盖到的分支为分序号的标准。

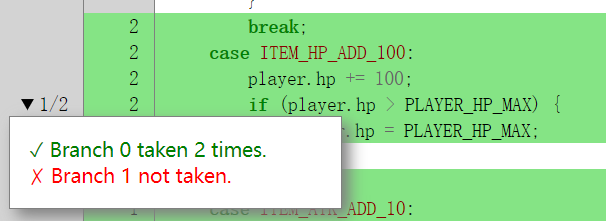
①：测试用例中没有出现inputBuf长度<1的情况。



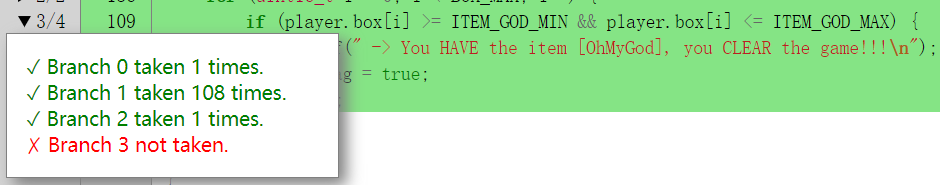
②：在使用道具的函数中，道具"ATK/2, but DEF+1"无法得到，对应的分支无法被覆盖。



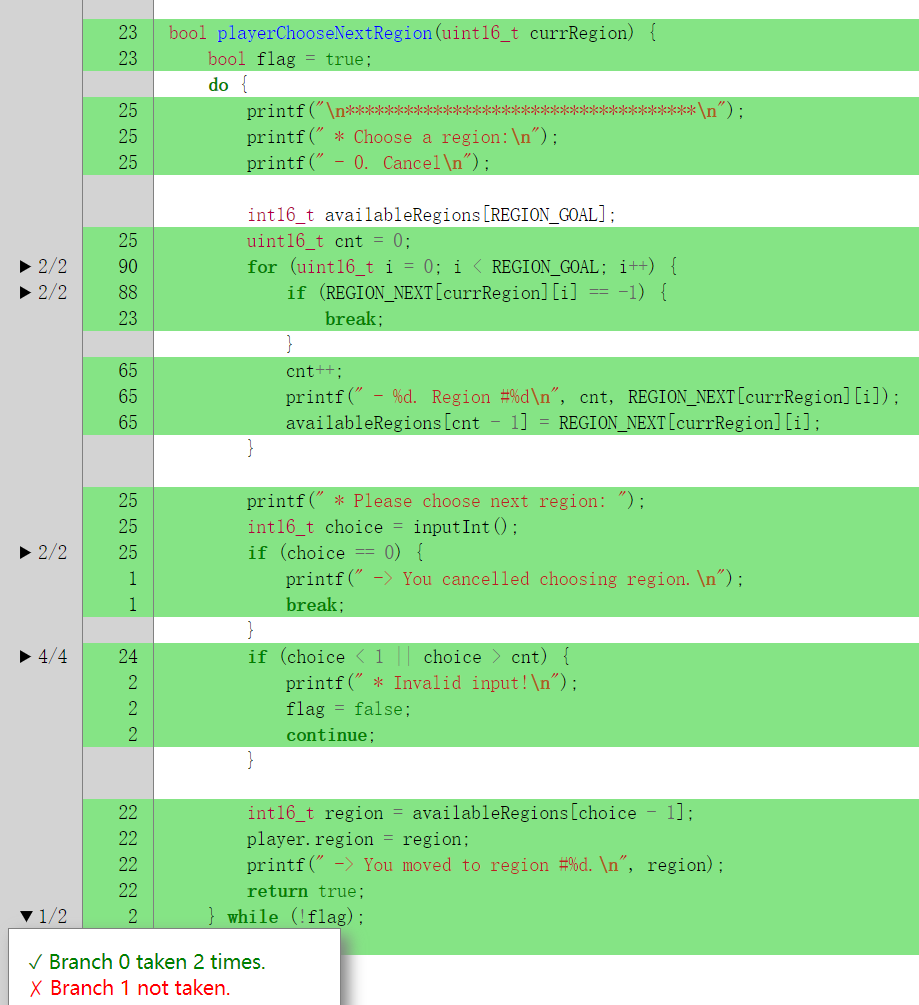
下图为使用道具HP+100的情况下，出现Branch 1 not taken的情况，因为玩家血量上限为100，而hp <= 0时游戏结束，所以使用HP+100时，player.hp > PLAYER\_HP\_MAX一定为真：



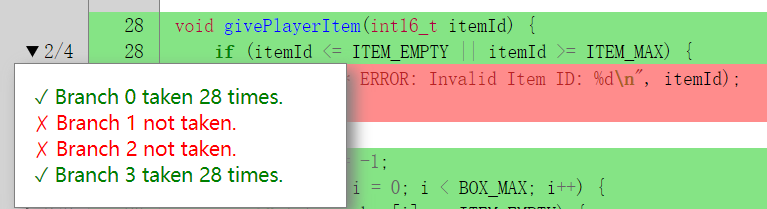
③：player.box[i] < ITEM\_GOD\_MIN的同时，player.box[i] > ITEM\_GOD\_MAX也发生的Branch3不会出现。



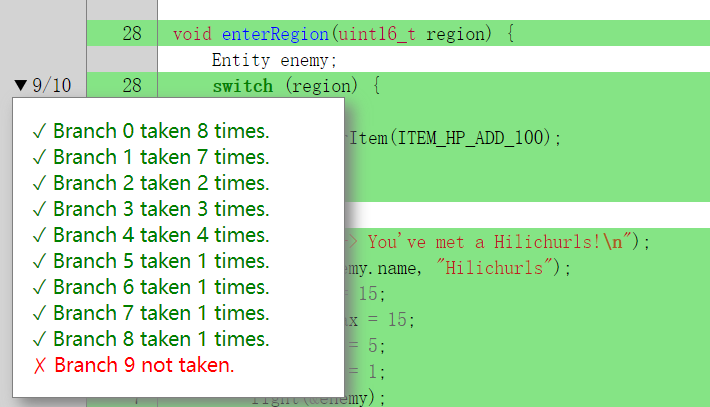
④：只有Invalid Input时才会进行判断，此时flag == false。

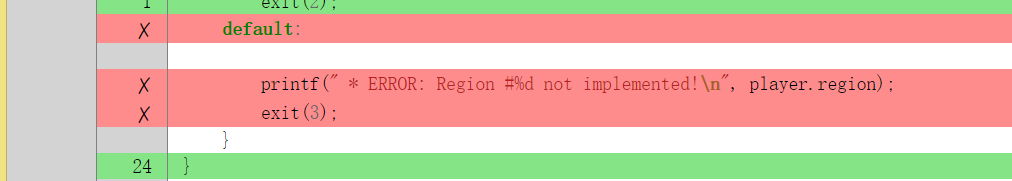


⑤：不会出现Invalid Item ID的错误情况

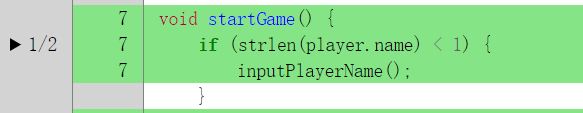


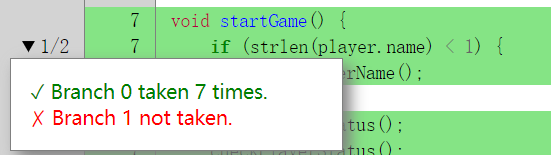
⑥：不会出现default的情况





⑦：输入player name时没有出现长度小于1的情况。





1. 总结

本次白盒测试测试用例的设计中，本人的设计思路是以提高语句覆盖率为目的，即在一个测试用例中增加测试内容，来尽可能多地覆盖到c文件中的语句。这样的设计思路是建立在本人对该测试程序比较了解，且测试程序的源代码内容相对较少的条件下的。尤其是后者，在源文件变得复杂后，以这种思路进行测试用例的设计的难度将会大大提升。在源文件变得复杂时，设计测试用例难免会出现对部分语句多次执行但其中部分执行是可以省掉的情况。而设计完一个类似input1.txt这样（在覆盖率这方面）相对而言十分优秀的测试用例后，再根据源文件和测试报告进行分析来补充测试用例。这样有了一个优秀的基础测试用例后再补充测试用例的思路受源文件复杂程度的影响较小。

在本次测试的过程中发现，白盒测试对于bug的发现上有一定的欠缺，比如本次测试用例中并没有黑盒测试中的几个bug测试，而在本次提交的测试用例基础上，增加例如DEF = 0被攻击后程序崩溃的测试用例后，对于gcov工具的报告中的覆盖率没有影响。因此，在测试中，使用黑盒和白盒相结合的测试方法会相比于使用单一的测试方法更有效。