种子杯 复赛试题

Judge Group

DIAN 启明学院

Contents

1. 题目概述 2
2. 基本要求 2
3. 提交规范 2
   3.1. 提交规范 2
   3.2. 提交时间 2
4. 评分说明 3
   4.1. 得分分数 3
      1. 分数分布 3
      2. 评分规则 3
   4.2. 测试说明 3
      1. 调用约定 3
      2. 输出格式 3
5. 详细需求 4
   5.1. 防止错误赋值（10’） 5
   5.2. 指针归空（10’） 5
   5.3. 无效分支（10’） 6
   5.4. 魔鬼数字（14’） 6
   5.5. 未使用的资源（16’） 7
      1. 函数 7
      2. 宏 8
      3. 变量 8
      4. 返回值 8
   5.6. 函数关系调用图（23’） 9
      1. 简单情况 9
      2. 递归调用 10
6. 参考 11

1. 题目概述

编译器的主要功能是将我们写的高级语言变成机器可以执行的指令，于是很多编程时候需要注意的问题编译器不会也没有必要帮我们检查。当然，类似于 VC 这样的 IDE 还是帮助我们做了很多的问题检查，例如一个变量是否没有被使用，指针类型有没有危险的强制转换。

但是，编译器能做的还是十分的有限。本次复赛就是对这样一些编译器没有做，但是确实有意义的一些功能点的一些尝试，例如：无效分支，资源泄露等。

我们检查的目标语言也是 C 语言。

2. 基本要求

1) 编程语言：C/C++ (使用库限制请参见 6 参考)
2) 运行平台：Windows Xp/vista/7/8
3) 编译器：VC/gcc (在 gcc 下编译，请确保代码的规范性，运行平台是 windows)

3. 提交规范

3.1. 提交规范

作品请以附件形式发送至大赛官方邮箱: seedcup@dian.org.cn。邮件主题命名方式：**复赛提交-队名。**要求将完整的工程和工程文档打包提交，压缩包命名规则为：**复赛提交-你们的队名.zip**，提交的目录格式如下：

```
─[复赛提交-你们的队名]
├─Src
├─Bin
└─Doc
```

目录说明如下：Src 文件夹放置源代码及工程文件；Bin 文件夹放置最后生成的可执行文件；Doc 放置说明文档。

文档中需要包含：

1) 程序的编译运行环境说明，包括使用的编辑器类型及版本。
2) 程序的设计结构及各模块的功能说明。
3) 对题目功能要求的完成情况，可以列举自己程序中觉得设计良好的部分，并说明为什么好。
4) 自定义的功能的详细描述与使用方法。
5) 你认为必要的附加信息，以方便评委了解你的想法。

请注意提交格式，严格按照提交格式提交你们的作品，提交格式不正确的我们酌情扣分处理。

3.2. 提交时间

比赛终止时间为 2012/11/11 晚上 10.00，请各位选手注意提交时间(以邮件发送时间为准)，我们不接受晚于终止时间提交的作品。

请选手做好提交准备，至少提前 5min 将作品投递到我们的邮箱，避免因为网络状况等原因导致无法按时提交的状况。

对于发件时间戳晚于 10.00 的队伍，我们将视为放弃比赛。

4. 评分说明

4.1. 得分分数

1. 分数分布

复赛总分为 110 分，其中 100 分在这份文档中间已经给出详细描述。

我们鼓励选手实现其它的附加功能，这些功能需要满足所描述的要求，我们会依据选手实现功能的难易与复杂程度来酌情附加 0~10 分。

分数项目	分值	评分标准
程序	83’	功能点的实现情况
代码	7’	编码的规范程度和代码的设计结构
文档	10’	逻辑和结构清晰，描述和图例详细，不得超过 10 页
其它功能	10’	选手实现的自定义功能

2. 评分规则

以测试用例的执行结果作为直接评分依据。以简单的测试用例为主，请重点实现给出的例子。

4.2. 测试说明

1. 调用约定

我们会如下调用选手提交的作品：

```
executable_file <input source filename> [-c] <output filename>
```

在命令行下面调用可执行文件的时候，后面将带有三个参数，这些参数的定义如下：

1. input source filename　　测试源代码的文件名字
2. output filename　　检查结果的输出文件
3. -c　　可选参数
   有-c 参数，则仅执行 5.6 的功能
   无-c 参数，则仅执行非 5.6 的全部功能

例如选手提交上来的作品名字为”seedpk.exe”，那么的调用的例子为：

```
seedpk.exe input.c output.txt
```

```
seedpk.exe in.c –c out.txt
```

2. 输出格式

函数关系调用图的输出格式请参见 5.6 具体功能点，其它测试点输出参见在此处的规定。例如，我们有

test.c 文件作为输入，内容如下：

```
1.    int func()
2.    {
3.        return 1;
4.    }
5.
6.    int main()
7.    {
8.        if(1>0)
9.        {
10.           func();
11.       };
12.       return;
13.   }
```

每次检测到一个错误，在输出文件中添加一行输出，输出顺序与在文件中出现顺序一致 （依据行号从小到大）。如果一行出现多个错误，则将在这一行出现的错误按照 error type 排序从小到大输出。

另外，我们的测试用例中一行一条语句，不会出现一条语句中途换行和多条语句在一行的情况。具体格式如下：

```
[line number=number],[error type=type]
```

两个参数用逗号分隔开来，并且它们之间是没有空格的。其中参数定义如下：

1. line number　　发生错误的行编号
2. error type　　错误类型
   错误类型总共有 5 大类，类型定义如下
   1.-- 5.1 防止错误赋值（10’）
   2.-- 5.2 指针归空（10’）
   3.-- 5.3 无效分支（10’）
   4.-- 5.4 魔鬼数字（14’）
   5.-- 5.5 未使用的资源（16’）

所以，在以上文件输入下，输出为：

```
line=8,error=3
line=10,error=5
```

如果该测试文件没有错误，则在创建的空文本文件中不用添加任何输出。

5. 详细需求

在复赛的测试用例中间，我们测试的代码都是命令行程序代码，均能编译通过（VS2008，VS2010，gcc），并且我们保证不会出现以下几种情况：

1. 指针类型的任意转换

2. 超过两层的指针和函数指针（int **p 允许，int ***p 不允许）
3. 条件编译（#define，#include 除外）
4. typedef 语句
5. 内存以外的系统资源
6. malloc/free/getchar 以外的系统库函数
7. structure 结构体类型和 enum 枚举类型的数据
8. 没有在测试文件中定义的宏（NULL 除外）
9. 超过 8 个字符的函数与变量名字

对于#include，在此特别说明，为保证测试代码的编译通过，所以测试代码中间会出现这条编译指令，但是选手不需要处理，可以直接忽略在测试代码中间出现的这条指令。

另外，我们用来测试的代码由于都已经通过 C 语言编译器编译，确定没有语法错误。所以，不用处理 C 语言语法错误的情况。

5.1. 防止错误赋值 (10’)

变量和常量做比较时，变量不能写在”==”符号的左边，以防止错误赋值。

```
1. #define TMPNUM 123
2. #define ANS 234
3. void main()
4. {
5.     int iTmp;
6.     iTmp = TMPNUM;
7.     if(iTmp == TMPNUM)
8.         iTmp = ANS;
9.     return;
10. }
```

这里第 7 行比较时将变量放在了左边，如果误将“==”写成“=”的话编译不会报错，但就形成了很难发现的逻辑错误。

在大型软件工程时为避免这种情况会规定将变量放在右侧比较，即：if(TMPNUM == iTmp)，这样误将“==”写成“=”时会直接编译报错。

```
line=7,error=1
```

5.2. 指针归空 (10’)

指针释放之后必须强制赋值为 NULL，以防止访问野指针。

```
1. #include<stdio.h>
2. #define TMP 5
3. int main()
4. {
5.     int *piVal = (int *)malloc(sizeof(int));
6.     *piVal = TMP;
```

```
7.     free(pival);
8.     return 0;
9.  }
```

这里第 7 行将 piVal 释放后并没有对其赋值为 NULL，应在第 7 行之后添加一句 piVal = NULL，并且应该在 free 指针这行报出警告。

```
line=7,error=2
```

5.3. 无效分支 (10’)

条件无效的分支是无效分支，减少无效分支从而减少冗余代码。

```
1.  #include <stdio.h>
2.  #define TWO 2
3.  int main()
4.  {
5.     int a = 0,b = 0;
6.     if(a>1)
7.        b=0;
8.     else if(a<1)
9.        b=getchar();
10.    else
11.       b=TWO;
12.    return 0;
13. }
```

第 6 行条件是永远不成立的，应在分支判断时报出警告。第 8 行条件是永远成立的，应在分支判断时报出警告。

```
line=6,error=3
line=8,error=3
```

5.4. 魔鬼数字 (14’)

在程序里面使用一个特定数字，但是这个数字的来历却毫无根据，这种数字就是魔鬼数字。魔鬼数字不方便阅读和程序的扩展。但是通常情况下，0 和正负 1 不属于“魔鬼数字”。

避免魔鬼数字通常利用宏定义数字。

```
1.  int main()
2.  {
3.    float fTemp = 1;
4.
5.    /* number should not be used here */
```

```
6.    fTemp = 10;
7.
8.    return 0;
9.  }
```

第 6 行中使用到了未说明来历的数字 10，因此需要在第 6 行应该发出错误警告。而第 3 行中的 1 属于常见数字，所以不用警告。

```
line=6,error=4
```

5.5. 未使用的资源 (16’)

在健壮的代码中，不应该出现申请了资源但是没有利用的情况。此检查项目分开为 4 类：

1) 未调用过的函数（main 除外）
2) 未使用过的宏
3) 未使用的变量
4) 未利用的函数返回值（如果函数返回值确实不必使用，则在调用的地方加上 void 声明）

1. 函数

如果一个函数有定义或者声明，但是在整个代码中间都无法调用到这个函数，那么这个函数就是没有被调用过的函数。

```
1.  #define FIVE 5
2.  int temp();
3.  int func();
4.
5.  int func()
6.  {
7.      return 1;
8.  }
9.
10. int main()
11. {
12.   int iTemp = 0;
13.   iTemp = FIVE;
14.   return 0;
15. }
```

第 2 行声明了函数 temp 但是既没有函数定义也没有被调用，所以属于没有使用过的函数，需要在声明的地方发出错误警告。

第 3 行有 func 函数的声明，第 5 行定义了函数 func。但是 func 在整个代码中没有被使用，那么在 func 定义的地方需要发出错误警告。

```
line=2,error=5
line=5,error=5
```

2. 宏

在代码中没有用到的宏是未使用的宏。

```
1. #define THREE 3
2. #define FOUR  4
3.
4. int main()
5. {
6.   int iTemp = 0;
7.
8.   iTemp = THREE;
9.
10.  return 0;
11.}
```

第 2 行中定义了宏 FOUR，但是在代码中没有使用这个宏，因此需要在宏定义的地方发出错误警报（不存在一个宏被多次重复定义的情况）。

```
line=2,error=5
```

3. 变量

如果一个被声明的变量，在它的整个作用域范围内没有被使用到，那么这个变量就是未使用的变量。对于未使用的变量，需要在它声明的位置发出错误警报。

```
1. int g_Temp = 0;
2.
3. int main()
4. {
5.   return 0;
6. }
```

全局变量 g_Temp 在代码中没有被使用，因此在 g_Temp 声明的地方需要发出错误警报。

```
line=1,error=5
```

4. 返回值

如果被调用的函数有返回值，但是在主调函数中却没有使用到这个返回值，那么就是函数返回值没有使用的错误。

```
1.  int func()
2.  {
3.    return 1;
4.  }
5.  void main()
6.  {
7.    (void)func();
8.    func();
9.  }
```

不需要函数返回值的时候，需要在函数调用的地方添加 void 标记，如第 7 行所示。第 8 行，调用同样的函数，却没有这个标记，那么就应该发出警告。

```
line=8,error=5
```

5.6. 函数关系调用图 (23’)

绘制函数调用关系图对理解大型程序大有帮助。

调用图的分析分析大致可分为“静态”和“动态”两种。静态分析是指在不运行待分析的程序的前提下进行分析，动态分析自然就是记录程序实际运行时的函数调用情况了。

在此，我们需要分析函数的静态调用关系。

1. 简单情况

我们需要将从 main 函数开始的所有函数的调用关系打印出来。

```
1. #include <stdio.h>
2. void func3()
3. {
4.     return;
5. }
6.
7. void func2()
8. {
9.     func3();
10. }
11.
12. void func1()
13. {
14.     func2();
15. }
16.
17. int main()
18. {
19.     func1();
```

```
20.    func1();
21.    func2();
22.    (void)getchar();
23.    return 0;
24.}
```

因此有以下输出：

```
main------func1-----func2-----func3
----------func2-----func3
----------getchar
```

注意：

1) 每个函数名占 10 个字符的位置，不足 10 个的用”-”补足 10 个
2) 在一个函数中，多次被调用的函数只用打印一次
3) 同一个主调函数中间的被调函数，从上向下按照 ANSI 顺序排列

2. 递归调用

```
1. void func1();
2.
3. void func2()
4. {
5.     func1();
6. }
7.
8. void func1()
9. {
10.    func2();
11.}
12.
13.int main()
14.{
15.    func1();
16.    return 0;
17.}
```

在这个例子中间，func1 调用到 func2，同时 func2 又调用到 func1，因此出现了循环的递归调用。对于这样的情况，我们需要打印到循环的起点为止，如下所示：

```
main------func1-----func2-----func1
```

6. 参考

1 使用库参考：http://en.cppreference.com/w/ （带有 C++ 11 标志是禁止的）

2 Google：http://www.google.com.hk

3 维基百科：http://en.wikipedia.org/wiki/Main_Page