实验报告

由于shell、malloc实验在操作系统课程中已经写过类似的，所以选择了data、cache、bomb三个实验。

不得不说csapp的前几个实验都很有意思，给了我和其他课程不同的实验体验，可惜并没能一一完成。总的来说这门课程像是一门沟通的桥梁，一定程度上兼容综合了本学期其他两门专业课（操作系统和组成原理），让我对系统知识有了更深刻的了解。

Data lab

实验目的

修改`bits.c`文件，使其满足实验要求。

1.实验步骤

由于事实上通过dlc测试和通过btest测试在本质上是相同的工作，这里不分开介绍。

bitXor(x,y)

功能：求两个数按位异或
由于只有两个数进行异或运算，因此只有当两个数分别为0和1时才为1，即两个数`or`计算为1但`and`不为1
因此我们可以用如下方式来表示

```
return (~((~x) & (~y))) & (~(x & y))
```

由于单目运算的括号可以去掉，可以简化为

```
return ~(~x & ~y) & ~(x & y)
```

tmin()

功能：返回补码最小的整数值
对于32位的int型，需要返回一个最高位为1其余为0的值，可以用如下方式：

```
return 1 << 31;
```

isTmax(x)

功能：如果x是以补码表示的最大的数则返回1反之为0
由于x是int型，其对应的最大值的补码形式为`0x7fff_ffff`，由于c语言并没有缩位运算，则需要对该数进行某种处理后使其正好为0之后再求非。可以用如下的代码得到0：

```
x = x + x + 1; // 0x7fff_ffff + (0x1000_0000)
x = ~x;
```

带几个特殊值之后发现-1的补码经过同样的计算后有相同的结果，因此考虑使用-1对应补码取反后为0的将其排除，但不幸的是`0x7fffffff`取反后也是0，因此使用+1为0的特性，且使用逻辑运算（否则仍无法分开），最终得到

```
int y = x + 1;
int z = ~(x + y);
int w = !y;   //防止0xffffffff
return !(z + w);
```

allOddBits(x)

说明：当奇数位均为1时返回1，反之为0
由于是奇数位均为1，考虑左移1位之后相加的全为1再取反，由于末尾补0，因此再+1，这个操作同样可以避免偶数位全为1。最终为了避免和为`0x80000000`，再做一次逻辑运算。代码实现如下：

```
int y = x << 1;
  y = ~(x + y + 1);
  return !y;
```

> 这里用btest测试后发现`0xffffffff`没有通过，经过查看后发现对题意理解有误orz，题目没有对偶数位进行要求，因此修改为

```
int mask = 0xAA | (0xAA << 8);
  mask = mask | (mask << 16);
  return !((mask & x) ^ mask);
```

negate(x)

要求返回参数的相反数，根据补码知识，很容易得出：

```
return ~x + 1;
```

isAsciiDigit(x)

说明：如果0x30 <= x <= 0x39（0-9对应ASCii值），返回1，否则返回0
根据题意，即通过位操作实现减法性质的操作。这里对两个边界做差取符号位即可。

```
    int mask = 1 << 31;
    int max = 0x39 + ~x + 1;
    int min = x + ~(0x30) + 1;
    max = (mask & max) >> 31;
    min = (mask & min) >> 31;
    return !(max|min);
```

conditional(x,y,z)

说明：实现`x ? y : z`
这里只需要通过一次逻辑运算实现？的功能，之后将逻辑值转换成对应的掩码`0x11111111`或`0x00000000`即可。代码实现如下：

```
x = ~(!x) + 1; //注意取非后是相反的，需要在return时体现
return (~x&y)|(x&z);
```

isLessOrEqual(x,y)

说明：如果x <= y返回1，否则为0
只需要通过补码做一次减法即可，在上面的`isAsciiDigit`部分已经实现过减法。

```
    int mask = 1 << 31;
    x = y + ~x + 1;
    return !((mask & x) >> 31);
```

> 经过btest测试后发现对于0和最小数的处理有疏漏，因此优先进行符号判断，修改为

```
int mask = 1 << 31;
  int cond1,cond2,cond3;
  cond2 = ((x >> 31) & 1) & !(y >> 31);
  cond3 = ((y >> 31) & 1) & !(x >> 31); //符号优先
  x = y + ~x + 1;
  cond1 = !((mask & x) >> 31);
  return (cond1 | cond2) & ~cond3;
```

logicalNeg(x)

说明：实现逻辑非
仅当为0时返回值为0，其余为1。只有0和`0x80000000`的取反 + 1是自己而不是对应的相反数，而`0x80000000`的最高位为1，因此只有0和他的取反+1最高位都是0，利用这个性质，可以得到：

```
    int mask = 1 ;
    return (~(x|~x + 1) >> 31 ) & mask;
```

特别注意，由于右移操作不改变符号，因此防止其余位的影响，最后应该用掩码1处理。

howManyBits(x)

- 未解决NaN的问题
  说明：计算一个数用补码最少需要几位表示
  从题面上看，这个函数的功能应该无法像前几个一样用较少的代码完成。由于必须要有一个符号位，因此只需要考虑其绝对值的情况，对于正数要找到符号位外从高到低第一个为1的位数，对于负数则需要找到第一个为0的位数，为了方便需要在一开始统一处理，将负数取反。

```
int mask = x >> 31; //利用了算数位移的特性
x = (mask & ~x) | (~mask & x);
```

为了尽可能的快速完成查找过程，利用一个类似于二分查找（大概叫这个？）的思路，实现如下：

```
int b0,b1,b2,b4,b8,b16;
b16 = !!(x >> 16) << 4;
x = x >> b16;
b8 = !!(x >> 8) << 3;
x = x >> b8;
b4 = !!(x >> 4) << 2;
x = x >> b4;
b2 = !!(x >> 2) << 1;
x = x >> b2;
b1 = !!(x >> 1);
x = x >> b1;
b0 = x;
return b0 + b1 + b2 + b4 + b8 + b16 +1;
```

- 该部分第一次在dlc测试中没有通过，报错`parse error`，后来通过尝试发现变量声明必须全放在最前面。

floatScale2(uf)

说明：参数和结果均为无符号整数，返回值为2*uf
由于需要考虑浮点数有可能是规格化和非规格化两种情况，再加上NaN，需要用到条件语句。

```
if((uf & 0x7f800000) == 0) //非规格化
    uf = ((uf & 0x007fffff) << 1) | (uf & 0x80000000);
else if((uf & 0x7f800000) != 0x7f800000) //规格化
    uf = uf + 0x00800000;
return uf;   //为NaN时原样返回
```

floatFloat2Int(uf)

说明：将单精度浮点数转换为对应的整数，NaN和无穷返回`0x80000000u`
由于参数和返回值均为无符号整型，同时浮点数和整数在负数的处理上并不同，需要进行转换，所以先用一个变量记录正负以及阶码和尾数：

```
int sign = uf >> 31;  //由于uf是无符号数，这里只可能是0或1
int exp = ((uf & 0x7f800000) >> 23) - 127;
int frac = (uf & 0x007fffff) | 0x00800000;  //需要加上1
```

由于浮点数转换为整数涉及到舍的问题，这里首先处理几个特殊情况
为防止出错，我们用一个这样的测试程序验证转换机制。

```
float y = 0.99;
int mask = y;
printf("%x %d\n",mask,mask);
```

结果如下，验证转换机制为直接舍去。

```
cattty@cattty-lap:~/csapp/data$ ./test
0 0
```

因此先处理一些特殊情况:

```
if(exp < 0)
    return 0;
if(!(uf & 0x7fffffff))
    return 0;
if(exp > 31) //NaN or infinite
    return 0x80000000;
```

此时将frac对齐：

```
if(exp > 23)
    frac = frac << (exp - 23);
else
    frac = frac >> (23 - exp);
```

最后把负数以及由于指数较大溢出的情况处理一下：

```
if(frac >> 31)
    return 0x80000000;
else if(sign)
    return ~frac + 1;
else
    return frac;
```

floatPower2(x)

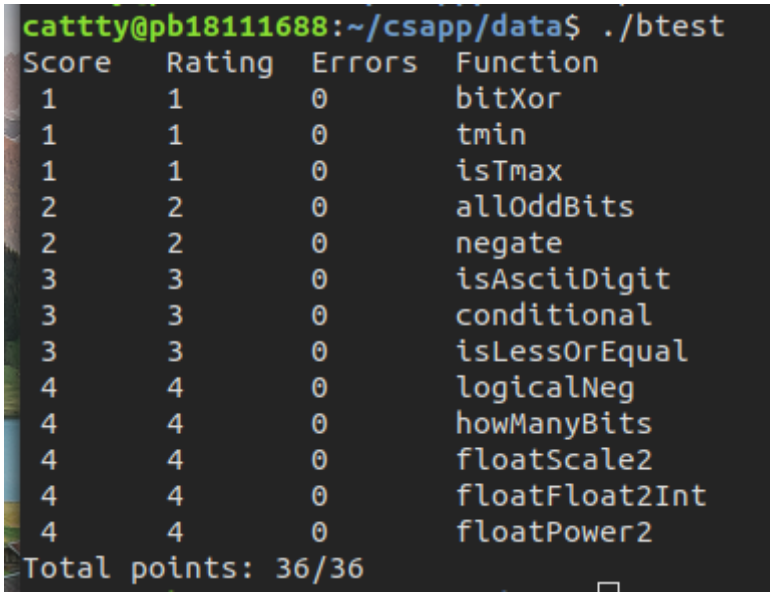
说明：计算2.0^x，返回的无符号整型实际上应该有符号位，如果结果太小，返回0，如果结果太大返回+infinite(`0x7f800000`)
该题目比较简单，直接按照规格化的单精度浮点数定义返回即可，由于规格化有一个默认的1，因此本题只需要移位。

测试

首先使用dlc进行测试，发现没有报错，说明没有违规代码

```
cattty@pb18111688:~/csapp/data$ ./dlc bits.c
cattty@pb18111688:~/csapp/data$
```

接下来使用btest测试，在这里发生了一个插曲，初次`make btest`时报错`cannot find -lgcc`，使用网上搜到的重新安装、镜像建立软连接等方式均无法解决，最后终于意识到`make btest`时使用的指令为`gcc -m32`，考虑可能由于32位的gcc存在库依赖缺失问题，执行`sudo aot-get install gcc-multilib`后解决问题。



```
cattty@pb18111688:~/csapp/data$ ./btest
Score   Rating  Errors  Function
 1      1       0       bitXor
 1      1       0       tmin
 1      1       0       isTmax
 2      2       0       allOddBits
 2      2       0       negate
 3      3       0       isAsciiDigit
 3      3       0       conditional
 3      3       0       isLessOrEqual
 4      4       0       logicalNeg
 4      4       0       howManyBits
 4      4       0       floatScale2
 4      4       0       floatFloat2Int
 4      4       0       floatPower2
Total points: 36/36
```

根据结果，所有测试全部通过。

cache lab

part A

这部分主要是要求写一个c程序模拟cache的运行，要求最后效果与`csim-ref`文件的执行效果一致。先看一下这个可执行文件大概的运行效果。
根据实验指导书，其命令行参数如下：

`Usage: ./csim-ref [-hv] -s <s> -E <E> -b <b> -t <tracefile>`

- -h: Optional help flag that prints usage info
- -v: Optional verbose flag that displays trace info
- -s <s>: Number of set index bits (S = 2s is the number of sets)
- -E <E>: Associativity (number of lines per set)
- -b <b>: Number of block bits (B = 2b is the block size)
- -t <tracefile>: Name of the valgrind trace to replay

用实验指导中给出的指令进行测试，效果如下：

```
cattty@pb18111688:~/csapp/cache$ ./csim-ref -s 4 -E 1 -b 4 -t traces/yi.trace
hits:4 misses:5 evictions:3
cattty@pb18111688:~/csapp/cache$ ./csim-ref -v -s 4 -E 1 -b 4 -t traces/yi.trace
L 10,1 miss
M 20,1 miss hit
L 22,1 hit
S 18,1 hit
L 110,1 miss eviction
L 210,1 miss eviction
M 12,1 miss eviction hit
hits:4 misses:5 evictions:3
```

下面开始试图用c程序编写实现。

实验过程

命令行解析

这个函数的首要任务就是对命令行进行处理，以实现命令行参数的功能，查找资料，可以使用命令行处理函数`getopt`，其原型为

```
nt getopt(int argc, char * const argv[], const char *optstring);
extern char *optarg;
```

这里我们可以用到的外部变量是`optarg`，含义为指向下一个参数的指针。对照getopt的一些使用例子，可以将命令行处理部分进行如下编写：

```
void cmdprocess(int argc, char* argv[])
{
    int opt;
    while ((opt = getopt(argc, argv, "hvs:E:b:t:")) != -1)
    {
        switch(opt){
            case 'h':
                fprintf(stdout, usage, argv[0]);
                exit(1);
            case 'v':
                verbose = 1;
                break;
            case 's':
                s = atoi(optarg);
                break;
            case 'E':
                E = atoi(optarg);
                break;
            case 'b':
                b = atoi(optarg);
                break;
            case 't':
                fp = fopen(optarg, "r");
                break;
```

```
            default:
                fprintf(stdout, usage, argv[0]);
                exit(1);
        }
    }
}
```

模拟cache

首先我们需要为cache定义一个数据结构，根据输出需要用到的有

```
typedef struct {
    int valid;  //有效位
    int tag;    //标记位
    int lru;    //least recently used
}cacheline;
typedef cacheline* cacheset;
typedef cacheSet* Cache;
```

定义好使用的数据结构之后,为cache开辟所需的空间。

```
int S = pow(2, s);
    cache = (Cache)malloc(sizeof(cacheSet) * S);
    if (cache == NULL) return -1;
    for (int i = 0; i < S; i++)
    {
        cache[i] = (cacheSet)calloc(E, sizeof(cacheLine));
        if (cache[i] == NULL) return -1;
    }
```

z这一步对应在编译的时候遇到了一个问题，即`pow`函数在执行时报错，查找了相关资料后发现是由于以下原因,添加`-lm`后解决

> 使用math.h中声明的库函数还有一点特殊之处，gcc命令行必须加-lm选项，因为数学函数位于libm.so库文件中（这些库文件通常位于/lib目录下），-lm选项告诉编译器，我们程序中用到的数学函数要到这个库文件里找。
>
> 下一步需要访问内存，这里我们可以将内存访问操作封装成一个函数方便使用。这里用注释进行说明。

```
int visitCache(int address)
{
    //首先先确定cache中查找时的序号和tag值，计算后在根据这个查找
    int tag = address >> (s + b);
    int setIndex = address >> b & ((1 << s) - 1);
    int evict = 0;
    int empty = -1;
    cacheSet cacheset = cache[setIndex];
```

```
    for (int i = 0; i < E; i++){
        if (cacheset[i].valid){
            if (cacheset[i].tag == tag){  //行匹配成功，hit++
                hits++;
                cacheset[i].lru = 1;
                return 0;
            }
            cacheset[i].lru++;
            if (cacheset[evict].lru <= cacheset[i].lru)
                evict = i;
        }
        else
            empty = i;
    }
    //处理miss的情况，由于hit的已经返回了
    //需要注意miss这里应该进行分类讨论
    misses++;
    cacheset[empty].tag = tag;
    cacheset[empty].lru = 1;
    if (empty != -1){
        cacheset[empty].valid = 1;
        return 1;
    }
    else{
        evictions++;
        return 2;
    }
}
```

完成对访问cache的封装后剩下的部分就比较简单了，主要就是一些对输入的标准化处理，使用sscanf相关的指令就可以解决。
编写开始的时候出现了一个小失误，即没有把`malloc`的内存`free`掉，通过其他的实验以及学习了解到这会发生内存泄漏，因此在最后需要`free(cache)`
根据实验要求，为了完成实验的A部分，最后需要调用`printSummary`，因此使用`printSummary(hits, misses, evictions);`
将需要的值输出，这里在第一次测试的时候报错，后来发现函数定义并没有包括在`cachelab.h`中，因此将其手动从`cachelab.c`中复制到`csim.c`中来。

测试

在实验指导书中给出的8个测试一一进行测试。

```
cattty@pb18111688:~/csapp/cache$ ./csim -s 1 -E 1 -b 1 -t traces/yi2.trace
hits:9 misses:8 evictions:6
cattty@pb18111688:~/csapp/cache$ ./csim -s 4 -E 2 -b 4 -t traces/yi.trace
hits:4 misses:5 evictions:2
cattty@pb18111688:~/csapp/cache$ ./csim -s 2 -E 1 -b 4 -t traces/dave.trace
hits:2 misses:3 evictions:1
cattty@pb18111688:~/csapp/cache$ ./csim -s 2 -E 1 -b 3 -t traces/trans.trace
hits:167 misses:71 evictions:67
cattty@pb18111688:~/csapp/cache$ ./csim -s 2 -E 2 -b 3 -t traces/trans.trace
hits:201 misses:37 evictions:29
cattty@pb18111688:~/csapp/cache$ ./csim -s 2 -E 4 -b 3 -t traces/trans.trace
hits:212 misses:26 evictions:10
cattty@pb18111688:~/csapp/cache$ ./csim -s 5 -E 1 -b 5 -t traces/trans.trace
hits:231 misses:7 evictions:0
cattty@pb18111688:~/csapp/cache$ ./csim -s 5 -E 1 -b 5 -t traces/long.trace
hits:265189 misses:21775 evictions:21743
cattty@pb18111688:~/csapp/cache$
```

```
cattty@pb18111688:~/csapp/cache$ ./csim-ref -s 1 -E 1 -b 1 -t traces/yi2.trace
hits:9 misses:8 evictions:6
cattty@pb18111688:~/csapp/cache$ ./csim-ref -s 4 -E 2 -b 4 -t traces/yi.trace
hits:4 misses:5 evictions:2
cattty@pb18111688:~/csapp/cache$ ./csim-ref -s 2 -E 1 -b 4 -t traces/dave.trace
hits:2 misses:3 evictions:1
cattty@pb18111688:~/csapp/cache$ ./csim-ref -s 2 -E 1 -b 3 -t traces/trans.trace
hits:167 misses:71 evictions:67
cattty@pb18111688:~/csapp/cache$ ./csim-ref -s 2 -E 2 -b 3 -t traces/trans.trace
hits:201 misses:37 evictions:29
cattty@pb18111688:~/csapp/cache$ ./csim-ref -s 2 -E 4 -b 3 -t traces/trans.trace
hits:212 misses:26 evictions:10
cattty@pb18111688:~/csapp/cache$ ./csim-ref -s 5 -E 1 -b 5 -t traces/trans.trace
hits:231 misses:7 evictions:0
cattty@pb18111688:~/csapp/cache$ ./csim-ref -s 5 -E 1 -b 5 -t traces/long.trace
hits:265189 misses:21775 evictions:21743
cattty@pb18111688:~/csapp/cache$
```

上下分别是按要求编写的程序以及参考程序的执行结果，对比测试结果，通过测试。

Bomb Lab

实验步骤

本实验的LICENSE部分看起来十分有趣，打开handout包之后并没有涵盖`bomb.c`需要include的头文件，而`bomb`文件也不是可查看类型。因此试图从反汇编的角度解题。由于之前未接触过反汇编，简单搜索后选择使用obidump工具
观察`.asm`文件并与c文件对应，可以认为实验共有6个phase，虽然asm文件中无法通过点击直接跳转调用的部分，但结合注释，观察其共同部分，都有形如

```
400ef0:   74 05                   je     400ef7 <phase_1+0x17>
400ef2:   e8 43 05 00 00          callq  40143a <explode_bomb>
```

即判断相等则跳转，否则执行`explode_bomb`，因此在各个阶段我们需要找到对应的字符串

phase1

```
0000000000400ee0 <phase_1>:
  400ee0: 48 83 ec 08           sub    $0x8,%rsp
  400ee4: be 00 24 40 00        mov    $0x402400,%esi
  400ee9: e8 4a 04 00 00        callq  401338 <strings_not_equal>
  400eee: 85 c0                 test   %eax,%eax
  400ef0: 74 05                 je     400ef7 <phase_1+0x17>
  400ef2: e8 43 05 00 00        callq  40143a <explode_bomb>
  400ef7: 48 83 c4 08           add    $0x8,%rsp
  400efb: c3                    retq
```

看到这里将一个地址给了`%esi`后执行了`string_not_equal`，猜测这个地址对应的就是phase1答案的地址，在gdb中进行调试：

```
cattty@pb18111688: ~/csapp/bomb
文件(F)  编辑(E)  查看(V)  搜索(S)  终端(T)  帮助(H)
(gdb) x /s 0x402400
0x402400:       "Border relations with Canada have never been better."
(gdb)
```

恰好为一个字符串`"Border relations with Canada have never been better."`，之前的猜想正确。

phase2

本阶段代码如下：

```
0000000000400efc <phase_2>:
  400efc:   55                      push   %rbp
  400efd:   53                      push   %rbx
  400efe:   48 83 ec 28             sub    $0x28,%rsp
  400f02:   48 89 e6                mov    %rsp,%rsi
  400f05:   e8 52 05 00 00          callq  40145c <read_six_numbers>
  400f0a:   83 3c 24 01             cmpl   $0x1,(%rsp)
  400f0e:   74 20                   je     400f30 <phase_2+0x34>
  400f10:   e8 25 05 00 00          callq  40143a <explode_bomb>
  400f15:   eb 19                   jmp    400f30 <phase_2+0x34>
  400f17:   8b 43 fc                mov    -0x4(%rbx),%eax
  400f1a:   01 c0                   add    %eax,%eax
  400f1c:   39 03                   cmp    %eax,(%rbx)
  400f1e:   74 05                   je     400f25 <phase_2+0x29>
  400f20:   e8 15 05 00 00          callq  40143a <explode_bomb>
  400f25:   48 83 c3 04             add    $0x4,%rbx
  400f29:   48 39 eb                cmp    %rbp,%rbx
  400f2c:   75 e9                   jne    400f17 <phase_2+0x1b>
  400f2e:   eb 0c                   jmp    400f3c <phase_2+0x40>
  400f30:   48 8d 5c 24 04          lea    0x4(%rsp),%rbx
  400f35:   48 8d 6c 24 18          lea    0x18(%rsp),%rbp
  400f3a:   eb db                   jmp    400f17 <phase_2+0x1b>
  400f3c:   48 83 c4 28             add    $0x28,%rsp
  400f40:   5b                      pop    %rbx
```

```
  400f41:   5d                      pop    %rbp
  400f42:   c3                      retq
```

在尚未仔细分析具体的汇编指令之前，观察几个函数名，通过`read_six_numbers`初步认为本阶段的字符串是由六个数字组成，本阶段没有显式的给出某个地址，因此只能逐步分析指令。
先进行化简，会触发`explode_bomb`的部分只有中间一段（后半段没有explode暂时先不考虑），即

```
  400f0a:   83 3c 24 01             cmpl   $0x1,(%rsp)
  400f0e:   74 20                   je     400f30 <phase_2+0x34>
  400f10:   e8 25 05 00 00          callq  40143a <explode_bomb>
  400f15:   eb 19                   jmp    400f30 <phase_2+0x34>
  400f17:   8b 43 fc                mov    -0x4(%rbx),%eax
  400f1a:   01 c0                   add    %eax,%eax
  400f1c:   39 03                   cmp    %eax,(%rbx)
  400f1e:   74 05                   je     400f25 <phase_2+0x29>
  400f20:   e8 15 05 00 00          callq  40143a <explode_bomb>
```

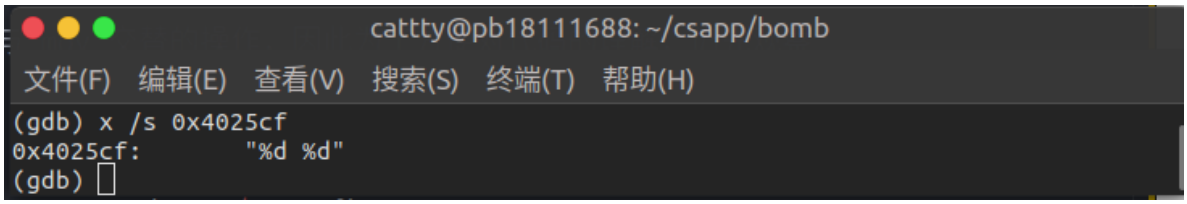
我们重点观察每次`explode_bomb`之前的比较，这里有效的信息是：先将`rsp`与1比较，再将`eax`赋值为`rbx`地址-4的内容后执行乘2操作，再与`rbx`进行比较。
查阅寄存器的相关知识，发现`rbx`地址-4的值与`rsp`相等，也就是第一个比较的值，由于后续没有`explode_bomb`，直观上认为这里是一个循环，由上面的分析结果，第一个数字是1，后面每个是前一个的两倍，因此这六个数字应该是`1 2 4 8 16 32`

phase3

本阶段汇编代码中有一大长串`jmp`与`mov`交替的操作，得知这里有一个switch函数，为了方便对代码的理解，仍先观察`explode_bomb`附近的指令。
第一次引爆前如下：

```
  400f43:   48 83 ec 18             sub    $0x18,%rsp
  400f47:   48 8d 4c 24 0c          lea    0xc(%rsp),%rcx
  400f4c:   48 8d 54 24 08          lea    0x8(%rsp),%rdx
  400f51:   be cf 25 40 00          mov    $0x4025cf,%esi
  400f56:   b8 00 00 00 00          mov    $0x0,%eax
  400f5b:   e8 90 fc ff ff          callq  400bf0
<__isoc99_sscanf@plt>
  400f60:   83 f8 01                cmp    $0x1,%eax
  400f63:   7f 05                   jg     400f6a <phase_3+0x27>
  400f65:   e8 d0 04 00 00          callq  40143a <explode_bomb>
```

这里我们既看到了一个地址，也看到了一个与1比较后引爆的操作，猜测phase3是对前两个阶段的一个综合，与阶段1一样，先查看在地址中存放的字符串



根据输出的结果，发现本阶段字符串是两个数字。接下来到switch段之间的代码如下：

```
   400f6a:   83 7c 24 08 07          cmpl   $0x7,0x8(%rsp)
   400f6f:   77 3c                   ja     400fad <phase_3+0x6a>
   400f71:   8b 44 24 08             mov    0x8(%rsp),%eax
   400f75:   ff 24 c5 70 24 40 00    jmpq   *0x402470(,%rax,8)
```

先将`rsp`+8中的值与7进行比较，如果大于7则跳转至`0x400fad`，而这里存放的是一条`explode_bomb`指令。下面将`rsp`+8的值放进`eax`然后对这个参数进行选择，结合之前判断与7的大小，知道下面这里是个8分支switch。
先观察最后的引爆段。

```
   400fbe:   3b 44 24 0c             cmp    0xc(%rsp),%eax
   400fc2:   74 05                   je     400fc9 <phase_3+0x86>
   400fc4:   e8 71 04 00 00          callq  40143a <explode_bomb>
```

得知这里是判断`rsp`+c，即第二个参数与`eax`是否相等，不相等则引爆，而eax是由上面的分支决定的，即前后两个数字对应就可以。以第一个参数为7为例，

```
   400fa6:   b8 47 01 00 00          mov    $0x147,%eax
   400fab:   eb 11                   jmp    400fbe <phase_3+0x7b>
```

这里我们就将第二个参数对应为`0x147`对应十进制类型为`327`，所以我们可以用`7 327`来解除炸弹。
同理，以下都是可以的答案：

```
0 207
1 311
2 707
3 256
4 389
5 206
6 682
```

phase4

与上一阶段一样，同样是先执行了一个`__isoc99_sscanf@plt`，使用gdb查看。在命令行中刚输入指令的时候发现所用地址和上一阶段是同一个，因此本阶段仍然是读取两个整数，分别放在`rcx`和`rdx`。
这里有一个调用`func4`，因此先看这之前的部分。

```
  401029:   83 f8 02                cmp    $0x2,%eax
  40102c:   75 07                   jne    401035 <phase_4+0x29>
  40102e:   83 7c 24 08 0e          cmpl   $0xe,0x8(%rsp)
  401033:   76 05                   jbe    40103a <phase_4+0x2e>
  401035:   e8 00 04 00 00          callq  40143a <explode_bomb>
  40103a:   ba 0e 00 00 00          mov    $0xe,%edx
  40103f:   be 00 00 00 00          mov    $0x0,%esi
  401044:   8b 7c 24 08             mov    0x8(%rsp),%edi
  401048:   e8 81 ff ff ff          callq  400fce <func4>
```

`explode_bomb`之前是对输入的检验，得知输入的两个数不能大于`0xe`，即14。下面对三个寄存器进行了赋值，应该是为了作为`func4`的参数。
在`func4`中存在递归调用，不方便直接分析，为了方便分析及测试用c语言表示：

```c
int func4(x, y, z){
  int a = x - y;
  int b = ((x - y) + z) >> 1;
  a = b + y;
  if(a > z){
    b = func4(a - 1, y, z);
    return 2 * b + 1;
  }
  else if(a < z){
    b = func4(x, a - 1, z);
    return 2 * b;
  }
  else
    return 0;
}
```

根据函数返回后的部分，得知当返回值为0时不会爆炸。

```
 40104d:    85 c0                  test   %eax,%eax
  40104f:   75 07                  jne    401058 <phase_4+0x4c>
```

这里手动分析较为复杂，编写简单的测试程序从0-14进行遍历，寻找可行值。结果如下

```
cattty@pb18111688:~/csapp/bomb$ ./test
0
1
3
7
```

所以第一个参数有4种可行取值。下面看第二个参数：

```
401051: 83 7c 24 0c 00          cmpl   $0x0,0xc(%rsp)
401056: 74 05                   je     40105d <phase_4+0x51>
401058: e8 dd 03 00 00          callq  40143a <explode_bomb>
```

很明显第二个参数只能为0，因此我们有`0 0`，`1 0`，`3 0`，`7 0`四种答案。

phase5

```
  401062:   53                      push   %rbx
  401063:   48 83 ec 20             sub    $0x20,%rsp
  401067:   48 89 fb                mov    %rdi,%rbx
  40106a:   64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
  401071:   00 00
  401073:   48 89 44 24 18          mov    %rax,0x18(%rsp)
  401078:   31 c0                   xor    %eax,%eax
  40107a:   e8 9c 02 00 00          callq  40131b <string_length>
  40107f:   83 f8 06                cmp    $0x6,%eax
  401082:   74 4e                   je     4010d2 <phase_5+0x70>
  401084:   e8 b1 03 00 00          callq  40143a <explode_bomb>
```

开始将输入放入`rbx`，从`string_length`附近我们得知本题答案是6个字符，来到跳转的`0x4010d2`位置，这里只是将`eax`清零后就跳转回来了。

```
  401089:   eb 47                   jmp    4010d2 <phase_5+0x70>
  40108b:   0f b6 0c 03             movzbl (%rbx,%rax,1),%ecx
  40108f:   88 0c 24                mov    %cl,(%rsp)
  401092:   48 8b 14 24             mov    (%rsp),%rdx
  401096:   83 e2 0f                and    $0xf,%edx
```

下面从栈里取了一个字节放进`ecx`，然后把上一部得到的`cl`中的值放进`rsp`、`rdx`中，接着取了这个字节的低四位放在`edx`中

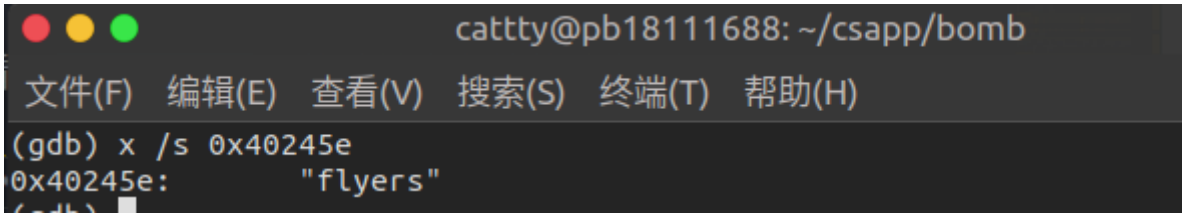
```
  401099:   0f b6 92 b0 24 40 00    movzbl 0x4024b0(%rdx),%edx
  4010a0:   88 54 04 10             mov    %dl,0x10(%rsp,%rax,1)
  4010a4:   48 83 c0 01             add    $0x1,%rax
  4010a8:   48 83 f8 06             cmp    $0x6,%rax
  4010ac:   75 dd                   jne    40108b <phase_5+0x29>
```

使用gdb，看到`0x4024b0`起的地址中存放的值为

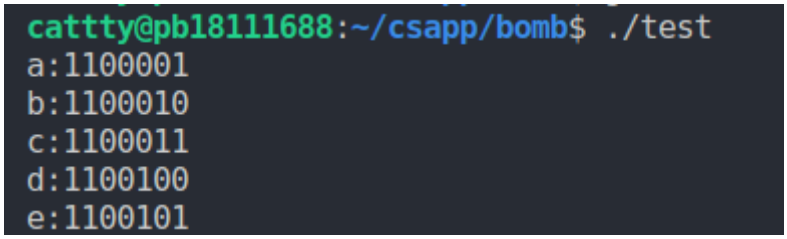
所以以`rdx`中的值为偏移量，取一个字节放进`edx`的低位，再将其放到栈里。这个过程循环6次。

```
4010ae:   c6 44 24 16 00          movb   $0x0,0x16(%rsp)
4010b3:   be 5e 24 40 00          mov    $0x40245e,%esi
4010b8:   48 8d 7c 24 10          lea    0x10(%rsp),%rdi
4010bd:   e8 76 02 00 00          callq  401338 <strings_not_equal>
4010c2:   85 c0                   test   %eax,%eax
4010c4:   74 13                   je     4010d9 <phase_5+0x77>
4010c6:   e8 6f 03 00 00          callq  40143a <explode_bomb>
```

这段很明显执行了一个字符比较的过程，开始的一行在最后加了字符串结束符。下面查看一下`0x40245e`中的内容：



所以我们需要找到这个字符串在上面那行里的位置，再将这个偏移量当做输入字符的低四位，就能获得字符串了，其中偏移量依次为`9, 15, 14, 5, 6, 7`，下面用一个测试程序获取一下字母和ascii的对应关系。



发现字符顺序正好是ascii二进制低四位，因此得到答案`ionefg`

phase6

从开始的`read_six_numbers`可以看出这个阶段仍然是6个数字组成的字符串。

```
40110b:   49 89 e6                mov    %rsp,%r14
40110e:   41 bc 00 00 00 00       mov    $0x0,%r12d
401114:   4c 89 ed                mov    %r13,%rbp
401117:   41 8b 45 00             mov    0x0(%r13),%eax
40111b:   83 e8 01                sub    $0x1,%eax
40111e:   83 f8 05                cmp    $0x5,%eax
401121:   76 05                   jbe    401128 <phase_6+0x34>
401123:   e8 12 03 00 00          callq  40143a <explode_bomb>
```

这一段说明输入的第一个数减1后需要小于等于5，即必须小于6，否则爆炸

```
401128:   41 83 c4 01             add    $0x1,%r12d
40112c:   41 83 fc 06             cmp    $0x6,%r12d
401130:   74 21                   je     401153 <phase_6+0x5f>
401132:   44 89 e3                mov    %r12d,%ebx
401135:   48 63 c3                movslq %ebx,%rax
401138:   8b 04 84                mov    (%rsp,%rax,4),%eax
40113b:   39 45 00                cmp    %eax,0x0(%rbp)
40113e:   75 05                   jne    401145 <phase_6+0x51>
401140:   e8 f5 02 00 00          callq  40143a <explode_bomb>
```

这里引爆前是比较了`eax`和`rbp`，往前看发现`eax`中是`rap`+4*`rax`的值，就是第`rax`个数字，从上一段可以看到`rbp`是`r13`指向的数字

```
401145:   83 c3 01                add    $0x1,%ebx
401148:   83 fb 05                cmp    $0x5,%ebx
40114b:   7e e8                   jle    401135 <phase_6+0x41>
40114d:   49 83 c5 04             add    $0x4,%r13
401151:   eb c1                   jmp    401114 <phase_6+0x20>
```

上面实在控制循环条件，`ebx`加1，`r13`指向下一个整数值，然后再返回循环，说明程序要求6个数都小于等于6且互不相等。

```
401153:   48 8d 74 24 18          lea    0x18(%rsp),%rsi
401158:   4c 89 f0                mov    %r14,%rax
40115b:   b9 07 00 00 00          mov    $0x7,%ecx
401160:   89 ca                   mov    %ecx,%edx
401162:   2b 10                   sub    (%rax),%edx
401164:   89 10                   mov    %edx,(%rax)
401166:   48 83 c0 04             add    $0x4,%rax
40116a:   48 39 f0                cmp    %rsi,%rax
40116d:   75 f1                   jne    401160 <phase_6+0x6c>
```

这里sub一步用`edx`减去`rax`，结合上面的mov，就是用7减去对应的数字存回原位，add用于不断移动当前处理的位置，cmp起到哨兵作用。

```
401197:   8b 0c 34                mov    (%rsp,%rsi,1),%ecx
40119a:   83 f9 01                cmp    $0x1,%ecx
40119d:   7e e4                   jle    401183 <phase_6+0x8f>
```

清零后跳转到以上的位置，把`rsp+rsi`的数据存进`ecx`，小于等于1则跳转。看一下如果跳转后在做什么

```
401183:   ba d0 32 60 00          mov    $0x6032d0,%edx
401188:   48 89 54 74 20          mov    %rdx,0x20(%rsp,%rsi,2)
40118d:   48 83 c6 04             add    $0x4,%rsi
401191:   48 83 fe 18             cmp    $0x18,%rsi
401195:   74 14                   je     4011ab <phase_6+0xb7>
```

第一行涉及到一个地址，用gdb看一下这里存的是什么，一开始使用了打印字符串的指令，但显示`<node1>`,提示我们这里有一个链表，因此将其全部打印出来，发现每个节点存了两个值，因此使用`x /12xg`命令，结果如下

将`rsp`+2*`rsi`+0x20的值赋给`rdx`而`rsi`每次增加4，所以每次都要存放8个字节的数据，过程重复6次，这里cmp一步有哨兵的作用。

```
401197:   8b 0c 34                mov    (%rsp,%rsi,1),%ecx
40119a:   83 f9 01                cmp    $0x1,%ecx
40119d:   7e e4                   jle    401183 <phase_6+0x8f>
```

将这个数赋给`ecx`，当`ecx`小于等于1时跳转
因此上这部分的功能是把链表里的值拿出来存到`rsp`+0x20为栈底的栈里。

```
4011ab:   48 8b 5c 24 20          mov    0x20(%rsp),%rbx
4011b0:   48 8d 44 24 28          lea    0x28(%rsp),%rax
4011b5:   48 8d 74 24 50          lea    0x50(%rsp),%rsi
4011ba:   48 89 d9                mov    %rbx,%rcx
4011bd:   48 8b 10                mov    (%rax),%rdx
```

这段主要的作用是把栈顶、第一个元素、栈底的位置分别存进了寄存器。

```
4011c0:   48 89 51 08             mov    %rdx,0x8(%rcx)
4011c4:   48 83 c0 08             add    $0x8,%rax
4011c8:   48 39 f0                cmp    %rsi,%rax
4011cb:   74 05                   je     4011d2 <phase_6+0xde>
4011cd:   48 89 d1                mov    %rdx,%rcx
4011d0:   eb eb                   jmp    4011bd <phase_6+0xc9>
4011d2:   48 c7 42 08 00 00 00    movq   $0x0,0x8(%rdx)
```

这里add相当于不断移动栈指针，将对应的数据存入并检查是否结束，最后一步在链表末尾加一个空节点
下面还需要搞清楚链表的顺序机制，由于过长只给出关键部分。

```
  4011e3:   8b 00                   mov    (%rax),%eax
  4011e5:   39 03                   cmp    %eax,(%rbx)
  4011e7:   7d 05                   jge    4011ee <phase_6+0xfa>
  4011e9:   e8 4c 02 00 00          callq  40143a <explode_bomb>
```

这里要求前一个节点的低4个字节值小于后一个节点，所以需要按照这个顺序对链表进行排序。根据前面gdb打出的结果，顺序为`3 4 5 6 1 2`，用7减去得到`4 3 2 1 6 5`

secret phase

查看phase6的时候偶然发现下面有`fun7`，结合c程序中的

> Wow, they got it! But isn't something... missing? Perhaps something they overlooked? Mua ha ha ha ha!
>
> 往下翻发现还有`secret phase`，结合c代码分析，应该是在`phase_defused`中调用的。所以直接看`phase_defused`部分，这部分有大量的可读性高的函数名，其中

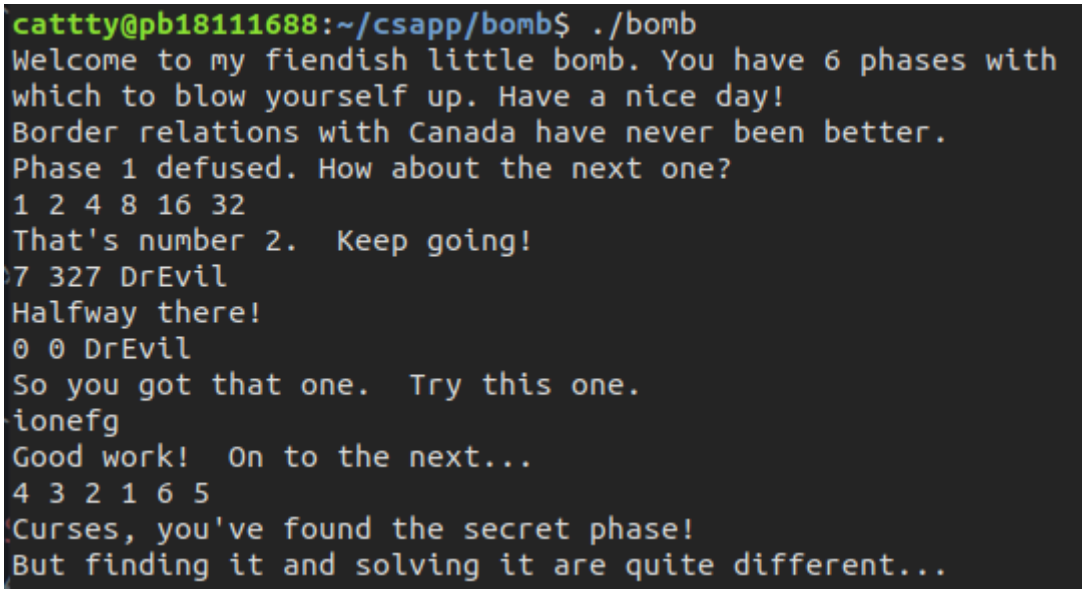
```
4015d8: 83 3d 81 21 20 00 06    cmpl   $0x6,0x202181(%rip)        #
603760 <num_input_strings>
4015df: 75 5e                   jne    40163f <phase_defused+0x7b>
```

将输入数字个数和6进行比较，为6时会执行中间的部分否则直接跳过。

```
  4015f0:   be 19 26 40 00          mov    $0x402619,%esi
  4015f5:   bf 70 38 60 00          mov    $0x603870,%edi
  4015fa:   e8 f1 f5 ff ff          callq  400bf0
<__isoc99_sscanf@plt>
  4015ff:   83 f8 03                cmp    $0x3,%eax
  401602:   75 31                   jne    401635
<phase_defused+0x71>
  401604:   be 22 26 40 00          mov    $0x402622,%esi
  401609:   48 8d 7c 24 10          lea    0x10(%rsp),%rdi
  40160e:   e8 25 fd ff ff          callq  401338 <strings_not_equal>
```

看一下格式化字符串的要求，这一段有多个可以查看的地方.

其中`0x603870`中什么都没有让人感到有些迷茫，因此先将DrEvil附加在可能的地方尝试一下。在执行3、4两步时都加上这个字符串：



根据结果，虽然不知道究竟是哪一步触发的，但该阶段确实可以被触发，因此继续研究该阶段，分析`secret_phase`部分。

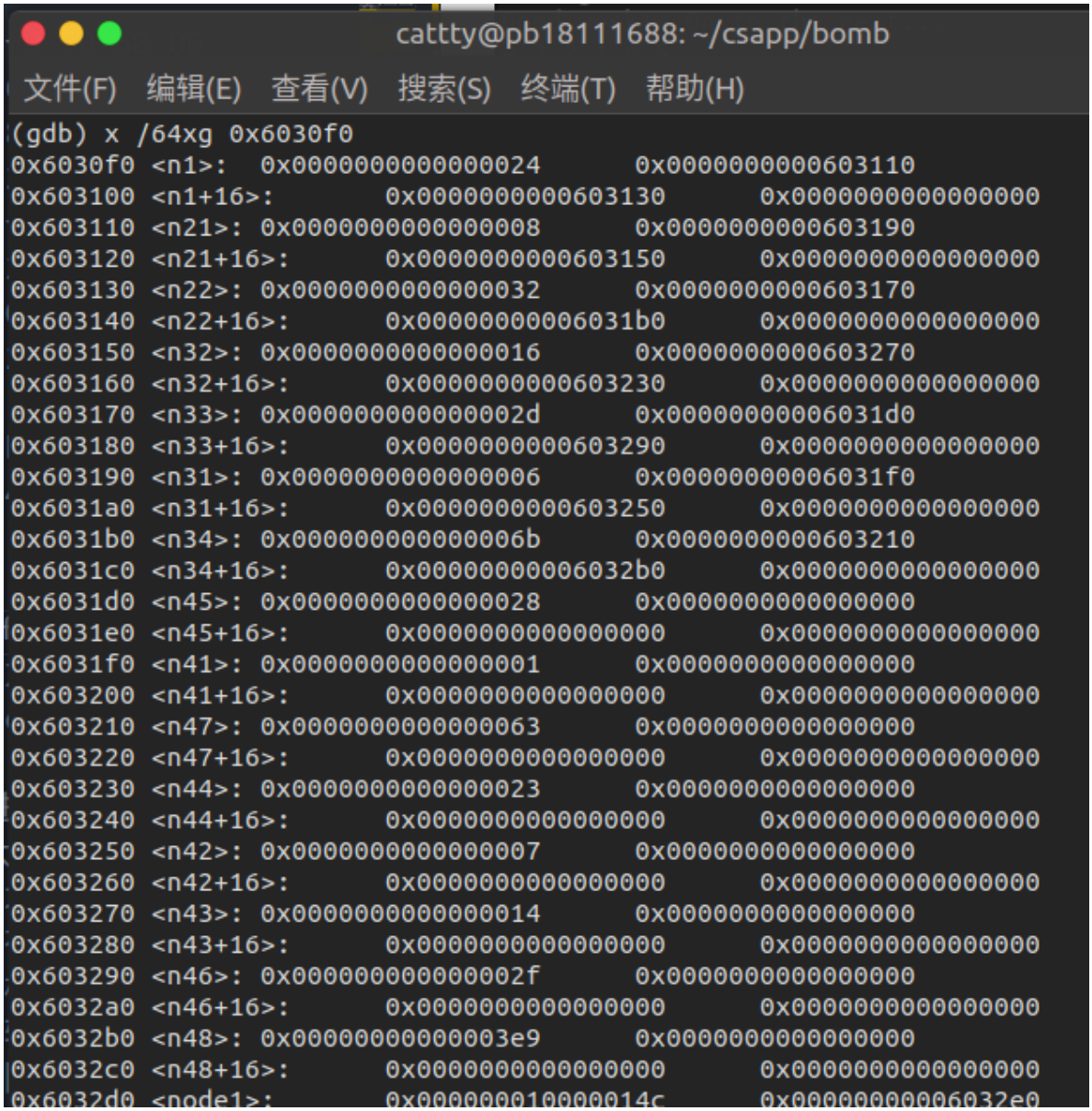
```
401255:   e8 76 f9 ff ff          callq  400bd0 <strtol@plt>
40125a:   48 89 c3                mov    %rax,%rbx
40125d:   8d 40 ff                lea    -0x1(%rax),%eax
401260:   3d e8 03 00 00          cmp    $0x3e8,%eax
```

这里发现还需要追加一行输入，而且是一个小于等于`0x3e8`，即1000的数。

```
40126e:   bf f0 30 60 00          mov    $0x6030f0,%edi
401273:   e8 8c ff ff ff          callq  401204 <fun7>
401278:   83 f8 02                cmp    $0x2,%eax
40127b:   74 05                   je     401282 <secret_phase+0x40>
40127d:   e8 b8 01 00 00          callq  40143a <explode_bomb>
```

其中`0x6030f0`是`$`,没看出来有什么用暂时认为是命令行提示符一类的东西，而cmp一句表示fun7的返回值需要是2，否则会爆炸。所以需要对`fun7`进行分析。

而`fun7`中大规模用到了`rdi`的值，也就是刚才的命令行提示符，因此认为这里有一个数据结构，所以换一种查看格式。经测试这个数据结构占据了较大的空间，多次尝试不同大小试图将其全部打出来。



经过观察这个应该是从低地址到高地址是以从上到下从左到右的层次遍历的方式表示的二叉树。

```
401208:   48 85 ff              test   %rdi,%rdi
40120b:   74 2b                 je     401238 <fun7+0x34>
40120d:   8b 17                 mov    (%rdi),%edx
40120f:   39 f2                 cmp    %esi,%edx
401211:   7e 0d                 jle    401220 <fun7+0x1c>
401213:   48 8b 7f 08           mov    0x8(%rdi),%rdi
401217:   e8 e8 ff ff ff        callq  401204 <fun7>
40121c:   01 c0                 add    %eax,%eax
40121e:   eb 1d                 jmp    40123d <fun7+0x39>
401220:   b8 00 00 00 00        mov    $0x0,%eax
401225:   39 f2                 cmp    %esi,%edx
401227:   74 14                 je     40123d <fun7+0x39>
401229:   48 8b 7f 10           mov    0x10(%rdi),%rdi
40122d:   e8 d2 ff ff ff        callq  401204 <fun7>
401232:   8d 44 00 01           lea    0x1(%rax,%rax,1),%eax
```

这部分实际上是根据二叉树节点值与输入值的比较情况返回不同的值，对应关系为：相等时返回0（最开始两行），输入值大时返回进入下一层同时返回`2*rax + 1`(最后一行代码),否则进入下一层并返回`2*rax`，为了获得2的结果，从最底层到顶层的返回结果应该是`0 -> 2*0+1=1 -> 2*1=2`,根据这个路径我们在树中寻找，应该取`0x16`，所以在隐藏阶段应该输入22.

测试

```
cattty@pb18111688:~/csapp/bomb$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Border relations with Canada have never been better.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2.  Keep going!
7 327 DrEvil
Halfway there!
0 0 DrEvil
So you got that one.  Try this one.
ionefg
Good work!  On to the next...
4 3 2 1 6 5
Curses, you've found the secret phase!
But finding it and solving it are quite different...
22
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!
```

结果显示通过全部测试。

实验总结

本次实验非常有意思而且代码量四舍五入就是没有，学习了通过反汇编辅助编程（虽然在这里并不是辅助作用）的方式。很好的练习了gdb调试以及内存查看的方法，通过翻译大量的汇编代码对一些寄存器名字更加熟悉。