一、以下是结构 test 的声明。假设在 32 位 Windows 平台上编译，问结构成员 d 和 v 的偏移量是多少？结构总大小是多少字节？如何调整成员的先后顺序使得结构所占存储空间最小？

```
struct {
    char c;
    int i;
    double d;
    short s;
    long l;
    void *v;
} test;
```

二、设无符号整型变量 ux 和 uy 的声明和初始化如下：

```
unsigned ux=x;
unsigned uy=y;
```

若 sizeof(int)=4，则对于任意 int 型变量 x 和 y，判断以下关系表达式是否永真，若永真则给出原理说明；若不永真则给出结果为假时 x 和 y 的取值。

(1) x/4+y/8==(x>>2)+(y>>3)

(2) x*4+y*8==(x<<2)+(y<<3)

三、遵循 Lab1 的规则要求，回答以下问题。

Lab1 的基本规则：只能使用顺序程序结构；仅能使用限定类型和数量的 C 语言算术和逻辑操作（详见各函数说明）；不得使用超过 8 位表示的常量、强制类型转换、数组/结构/联合等数据类型、宏等；不得定义或调用其他函数等。

函数名	功能	约束条件	最多操作符数量
int logicalOr(int x, int y)	如果 x 和 y 都等于 0 则返回 0，否则都返回 1。	仅能使用 ！ ~ & ^ | + << >>	20
int isPositive(int x)	如果 x 大于 0 返回 1，否则返回 0	仅能使用 ！ ~ & ^ | + << >>	8

(1) 写出函数 logicalOr 的实现

```
int logicalOr (int x, int y)
{


}
```

(2) 写出函数 isPositive 的实现

```
int isPositive (int x)
{


}
```

四、已知函数 func 的 C 语言代码及其对应的汇编代码如下所示：

```
#include<stdio.h>
int func(void) {
   int  x, y;
   scanf (“%d %d”, &x, &y);
   return  x-y;
}
```

```
    func:
1   pushl   %ebp
2   movl    %esp, %ebp
3   subl    $40, %esp
4   leal    -8(%ebp), %eax
5   movl    %eax, 8(%esp)
6   leal    -4(%ebp), %eax
7   movl    %eax, 4(%esp)
8   movl    $.LC0, (%esp)          # 将指向字符串“%d %d”的指针入栈
9   call    scanf
10  movl    -4(%ebp), %eax
11  subl    -8(%ebp), %eax
12  learve
13  ret
```

假设函数 func 开始执行时（指第 2 行指令执行前的时刻），R[esp]=0xbc000020, R[ebp]=0xbc000030, 执行第 10 行 call 指令后，scanf 从标准输入读入的值为 15 和 20，指向字符串“%d %d”的指针为 0x804c000。

回答下列问题。

(1)从开始执行第 2 行指令到执行完第 13 行指令，分析这个过程中寄存器 EBP 和 ESP 的内容变化情况。

(2)局部变量 x 和 y 所在的存储单元地址分别是什么？

(3)画出执行第 10 行指令后 func 的栈帧，指出栈帧中的内容及其地址。

五、缓冲区溢出攻击实验（lab3）的 bang 阶段：某次执行时，有以下操作和相关输出信息：

注：

```
zhanglx:~/lab3$ gdb bufbomb
GNU gdb (Ubuntu 7.12.50.20170314-0ubuntu1.1) 7.12.50.20170314-git
Copyright (C) 2017 Free Software Foundation, Inc.
......
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from bufbomb...(no debugging symbols found)...done.
(gdb) b getbuf
Breakpoint 1 at 0x80491f2
(gdb) r -u U201514778
Starting program: /home/zhanglx/lab3/bufbomb -u U201514778
Userid: U201514778
Cookie: 0x4b8aa076

Breakpoint 1, 0x080491f2 in getbuf ()
(gdb) i r
eax            0x25dea32d  635347757
ecx            0x25dea32d  635347757
edx            0xb7fb63e4  -1208261660
ebx            0x0    0
esp            0x55683378  0x55683378 <_reserved+1037176>
ebp            0x556833b0  0x556833b0 <_reserved+1037232>
......

(gdb) x /20x 0x0804c218
0x804c218 <global_value>:  0x00000000    0x00000000    0x4b8aa076    0xb7fb65a0
......
```

（1）上述显示的信息里，“……”为省略部分，省略的内容对解答本题没有影响（下同）。

（2）设存储分配时全局变量 cookie 分配在 global_value 之后，且利用上面的信息可以唯一确定 cookie 的存储位置。

以下是相关的 C 语言源程序：

```
.........
/* Buffer size for getbuf */
#define NORMAL_BUFFER_SIZE 32
int getbuf()
{
    char buf[NORMAL_BUFFER_SIZE];
    Gets(buf);
    return 1; //正常时返回1
}
......

/* $begin bang-c */
int global_value = 0;
void bang(int val)
{
    if (global_value == cookie) {
        printf("Bang!: You set global_value to 0x%x\n", global_value);
        validate(2);
    } else
        printf("Misfire: global_value = 0x%x\n", global_value);
    exit(0);
}
/* $end bang-c */
```

以下是 objdump –d bufbomb 输出的部分信息：

```
080491ec <getbuf>:
 80491ec:    55                   push   %ebp
 80491ed:    89 e5                mov    %esp,%ebp
 80491ef:    83 ec 38             sub    $0x38,%esp
 80491f2:    8d 45 d8             lea    -0x28(%ebp),%eax
 80491f5:    89 04 24             mov    %eax,(%esp)
 80491f8:    e8 55 fb ff ff       call   8048d52 <Gets>
 80491fd:    b8 01 00 00 00       mov    $0x1,%eax
 8049202:    c9                   leave
 8049203:    c3                   ret

......
08048d05 <bang>:
 8048d05:    55                   push   %ebp
 8048d06:    89 e5                mov    %esp,%ebp
 8048d08:    83 ec 18             sub    $0x18,%esp
......
```

基于上面的已知条件和信息，回答下面的问题：

（1）在当前执行状态下，getbuf 函数中的缓冲区 buf 的首地址是多少？写出必要的分析和计算过程。

（2）某同学构造了一个攻击字符串，如下所示，请完善空缺部分（必须用十六进制表示，每个字节用两个十六进制数表示）。

00 00 00 00 a1 __ __ __ __ a3 __ __ __ __
c3 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 __ __ __ __ __ __ __ __

六、下表给出的是某个可执行目标文件程序头表的部分信息。

Type	Offset	VirtAddr	PhysAddr	FileSiz	MenSiz	Flg	Align
……							
LOAD	0x00000000	0x08048000	0x08048000	0x00000448	0x00000448	RE	0x1000
LOAD	0x00000448	0x08049448	0x08049448	0x000000e8	0x00000104	RW	0x1000
……							

根据表中的信息，回答问题：可读写数据段在虚拟存储空间中的起始地址为：（①），长度为（②）个字节，其数据来自可执行文件中偏移地址为（③）开始的（④）个字节。

由此可见，可执行目标文件中的数据长度和虚拟地址空间中的存储区大小之间相差了（⑤）个字节，解释可能的原因。

①_____

②_____

③_____

④_____

⑤_____

可能的原因是：

七、设一程序由两个模块 test.c 和 app.c 组成，其中 test.c 中包含 main 函数，app.c 中包含 swap 和 output 两个函数。对两个模块单独编译，分别生成它们对应的可重定位目标文件 test.o 和 app.o。test.c 和 swap.c 的源程序、test.o 的反汇编（objdump –d test.o）输出如下所示。

假设链接后生成可执行目标文件 t，并假定可执行目标文件里 main 函数代码的起始地址是 0x8048386，而紧跟在 main 函数后的是 swap 函数的代码。设函数首地址按 4 字节边界对齐。请回答后面的问题

main.c

```
extern void swap(void);
extern void output(void);
int buf[2]={1,-2};
int sum;
int main()
{
    swap();
    sum = buf[0] + buf[1];
    output();
    return 0;
}
```

swap.c

```
#include <stdio.h>
extern int buf[];
int *bufp0 = &buf[0];
static int *bufp1;
long long sum = 0;
void swap()
{
    int temp;
    bufp1 = & buf[1];
    temp = *bufp0;
    *bufp0 = * bufp1;
    *bufp1 = temp;
}
void output()
{
    printf("%lld\n",sum);
}
```

objdump -d test.o

```
00000000 <main>:
   0:    55                      push   %ebp
   1:    89 e5                   mov    %esp,%ebp
   3:    83 ec 08                sub    $0x8,%esp
   6:    e8 fc ff ff ff          call   7 <call+0x7>     # 重定位位置①
                      7: ……
   b:    8b 15 00 00 00 00       mov    0x0,%edx         # 重定位位置②
                      d: ……
  11:    a1 04 00 00 00          mov    0x4,%eax         # 重定位位置③
                      12: ……
  16:    01 d0                   add    %edx,%eax
  18:    a3 00 00 00 00          mov    %eax,0x0         # 重定位位置④
                      19: R_386_32  sum
  1d:    e8 fc ff ff ff          call   1e <call+0x1e>   # 重定位位置⑤
                      1e: R_386_PC32    output
  22:    b8 00 00 00 00          mov    $0x0,%eax
  27:    c9                      leave
  28:    c3                      ret
```

gdb t 的部分输出：

```
zhanglx:~/program$ gdb t
GNU gdb (Ubuntu 7.12.50.20170314-0ubuntu1.1) 7.12.50.20170314-git
Copyright (C) 2017 Free Software Foundation, Inc.
……
Reading symbols from t1...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x8048475
(gdb) r
Starting program: /home/zhanglx/program/t

Breakpoint 1, 0x08048475 in main ()
(gdb) x /20x 0x0804a020
0x804a020 <buf>:  0x00000001    0xfffffffe    0x00000000    0x00000000
0x804a030 <sum>:  0x00000000    0x00000000    0x00000000    0x00000000
……
```

问题：

（1）对 test.o 中的重定位位置①进行重定位。回答：此处待重定位的符号是什么、重定位类型是什么？重定位前的值及该值的含义是什么、重定位后的值是什么（需要给出计算过程）及重定位后最终指向的虚拟地址是多少。

（2）以下是执行该程序时的输出信息。请解释输出此结果的原因。

```
zhanglx:~/program$ ./t
4294967295
```