

# 哈尔滨工业大学

# 实验报告

## 实 验（二）

题 目 DataLab 数据表示

专 业 计算学部

学 号 1190200708

班 级 1903008

学 生 熊峰

指 导 教 师 吴锐

实 验 地 点 G709

实 验 日 期 2021.3.29

## 计算机科学与技术学院

# 目 录

<b>第 1 章 实验基本信息 .....</b>	<b>- 4 -</b>
1.1 实验目的.....	- 4 -
1.2 实验环境与工具.....	- 4 -
1.2.1 硬件环境.....	- 4 -
1.2.2 软件环境.....	- 4 -
1.2.3 开发工具.....	- 4 -
1.3 实验预习.....	- 4 -
<b>第 2 章 实验环境建立 .....</b>	<b>- 6 -</b>
2.1 UBUNTU 下 CODEBLOCKS 安装.....	- 6 -
2.2 64 位 UBUNTU 下 32 位运行环境建立.....	- 7 -
<b>第 3 章 C 语言的数据类型与存储 .....</b>	<b>- 8 -</b>
3.1 类型本质（1 分） .....	- 8 -
3.2 数据的位置-地址（2 分） .....	- 8 -
3.3 MAIN 的参数分析（2 分） .....	- 11 -
3.4 指针与字符串的区别（2 分） .....	- 12 -
<b>第 4 章 深入分析 UTF-8 编码.....</b>	<b>- 13 -</b>
4.1 提交 UTF8LEN.C 子程序.....	- 13 -
4.2 C 语言的 STRCMP 函数分析 .....	- 14 -
4.3 讨论：按照姓氏笔画排序的方法实现 .....	- 14 -
<b>第 5 章 数据变换与输入输出 .....</b>	<b>- 15 -</b>
5.1 提交 CS_ATOI.C .....	- 15 -
5.2 提交 CS_ATOF.C .....	- 15 -
5.3 提交 CS_ITOA.C .....	- 15 -
5.4 提交 CS_FTOA.C .....	- 15 -
5.5 讨论分析 OS 的函数对输入输出的数据有类型要求吗 .....	- 15 -
<b>第 6 章 整数表示与运算 .....</b>	<b>- 17 -</b>
6.1 提交 FIB_DG.C.....	- 17 -
6.2 提交 FIB_LOOP.C .....	- 17 -
6.3 FIB 溢出验证.....	- 17 -
6.4 除以 0 验证： .....	- 17 -
<b>第 7 章 浮点数据的表示与运算 .....</b>	<b>- 19 -</b>
7.1 正数表示范围.....	- 19 -
7.2 浮点数的编码计算 .....	- 19 -

7.3 特殊浮点数值编码 .....	- 20 -
7.4 浮点数除 0 .....	- 20 -
7.5 FLOAT 的微观与宏观世界 .....	- 20 -
7.6 讨论：任意两个浮点数的大小比较 .....	- 21 -
<b>第 8 章 舍位平衡的讨论 .....</b>	<b>- 22 -</b>
8.1 描述可能出现的问题 .....	- 22 -
8.2 给出完美的解决方案 .....	- 22 -
<b>第 9 章 总结 .....</b>	<b>- 23 -</b>
9.1 请总结本次实验的收获 .....	- 23 -
9.2 请给出对本次实验内容的建议 .....	- 23 -
<b>参考文献 .....</b>	<b>- 24 -</b>

## 第 1 章 实验基本信息

### 1.1 实验目的

熟练掌握计算机系统的数据表示与数据运算  
通过 C 程序深入理解计算机运算器的底层实现与优化  
掌握 VS/CB/GCC 等工具的使用技巧与注意事项

### 1.2 实验环境与工具

#### 1.2.1 硬件环境

X86-64 CPU; 3.60GHz; 16G RAM; 256G SSD; 1T SSD

#### 1.2.2 软件环境

Win 10

Ubuntu 20.04.2 LTS

#### 1.2.3 开发工具

Visual Studio 2019; Vim; GCC; GDB; Code::Blocks; CLion 2020.3.1 x64

### 1.3 实验预习

上实验课前，必须认真预习实验指导书（PPT 或 PDF）

了解实验的目的、实验环境与软硬件工具、实验操作步骤，复习与实验有关的理论知识。

采用 sizeof 在 Windows 的 VS/CB 以及 Linux 的 CB/GCC 下获得 C 语言每一类型在 32/64 位模式下的空间大小

Char /short int/int/long/float/double/long long/long double/指针

编写 C 程序，计算斐波那契数列在 int/long/unsigned int/unsigned long 类型时，n 为多少时会出错

先用递归程序实现，会出现什么问题？

再用循环方式实现。

写出 `float/double` 类型最小的正数、最大的正数（非无穷）

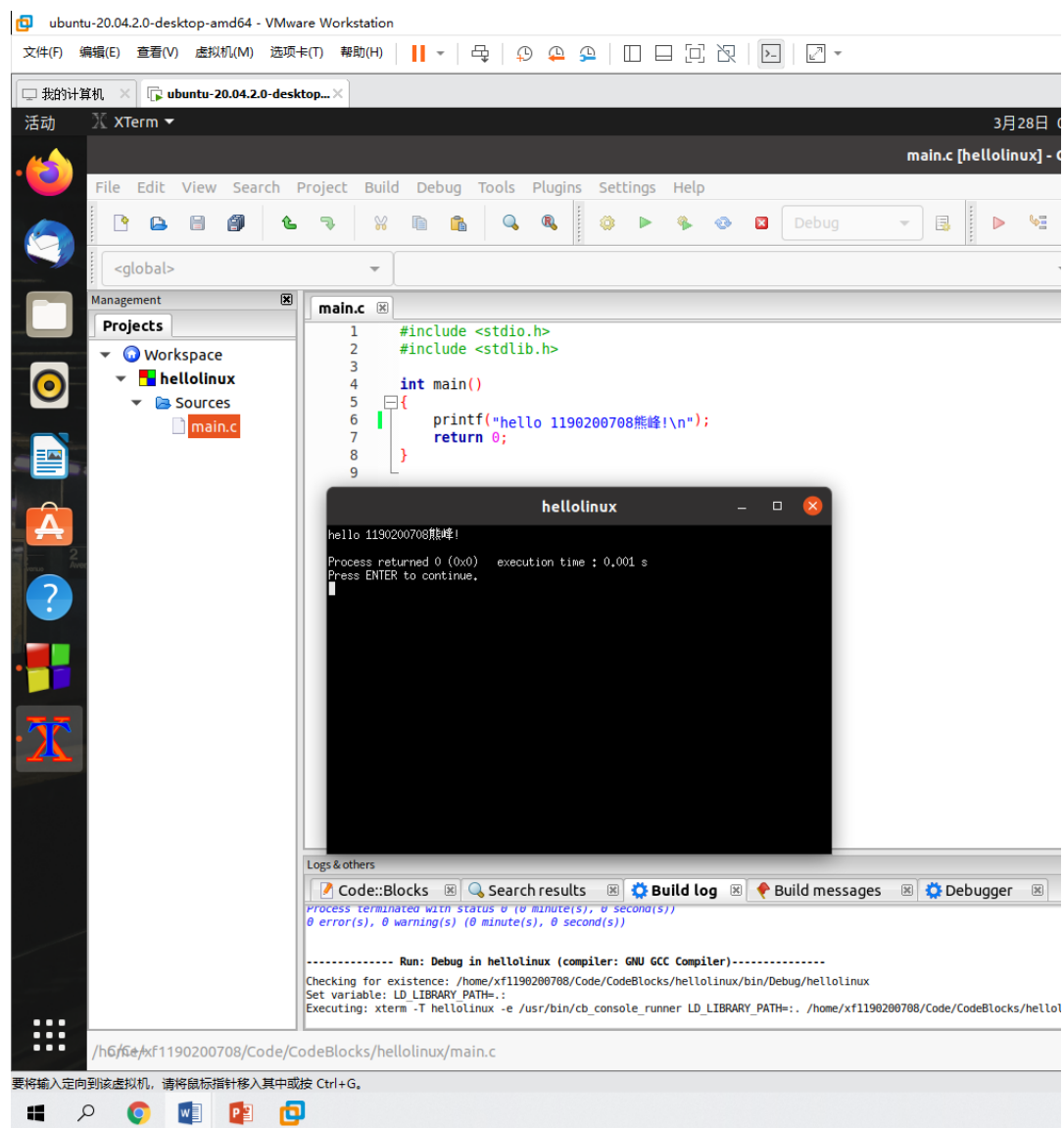
按步骤写出 `float` 数-1.1 在内存从低到高地址的字节值-16 进制

按照阶码区域写出 `float` 的最大密度区域范围及其密度，最小密度区域及其密度（区域长度/表示的浮点个数）

## 第 2 章 实验环境建立

### 2.1 Ubuntu 下 CodeBlocks 安装

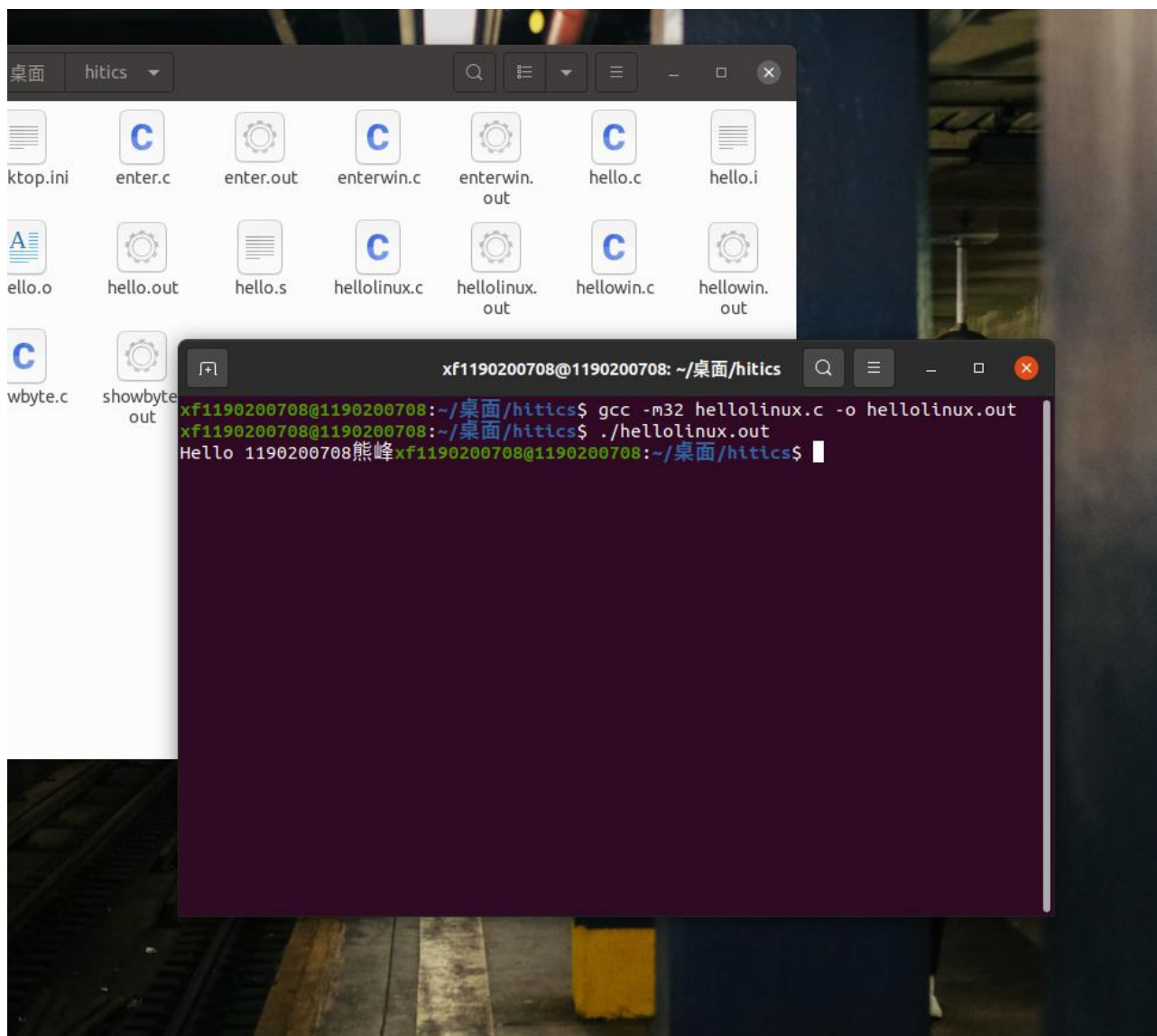
CodeBlocks 运行界面截图：编译、运行 hellolinux.c



## 2.2 64 位 Ubuntu 下 32 位运行环境建立

在终端下，用 gcc 的 32 位模式编译生成 hellolinux.c。执行此文件。

Linux 及终端的截图。



## 第 3 章 C 语言的数据类型与存储

### 3.1 类型本质 (1 分)

	Win/VS/x86	Win/VS/x64	Win/CB/32	Win/CB/64	Linux/CB/32	Linux/CB/64
char	1	1	1	1	1	1
short	2	2	2	2	2	2
int	4	4	4	4	4	4
long	4	4	4	8	4	8
long long	8	8	8	8	8	8
float	4	4	4	4	4	4
double	8	8	8	8	8	8
long double	8	8	12	16	12	16
指针	4	8	4	8	4	8

C 编译器对 sizeof 的实现方式：C/C++ 中，sizeof() 只是运算符并非函数，在编译时会被编译器直接替换为常量，即使是汇编代码都只能看到一个常量。

Eg: 图中为 sizeof() 运算符生成的汇编代码，可以观察到，此时 sizeof(a) 已经被转化为立即数 4。

```

unsigned sizeofint()
{
    return sizeof(int);
}

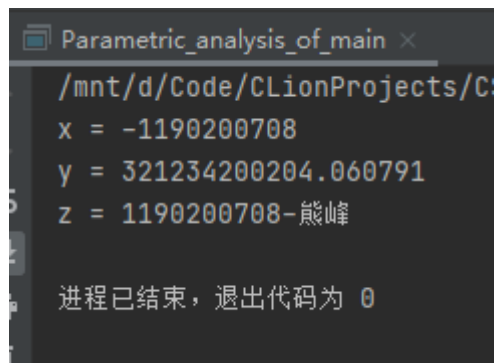
int main()
{
    unsigned x = sizeofint();
    return 0;
}
0x000055555555129 <+0>: endbr64
0x00005555555512d <+4>: push    %rbp
0x00005555555512e <+5>: mov     %rsp,%rbp
0x000055555555131 <+8>: mov     $0x4,%eax
0x000055555555136 <+13>: pop     %rbp
0x000055555555137 <+14>: retq

```

### 3.2 数据的位置-地址 (2 分)

打印 x、y、z 输出的值：截图 1





```

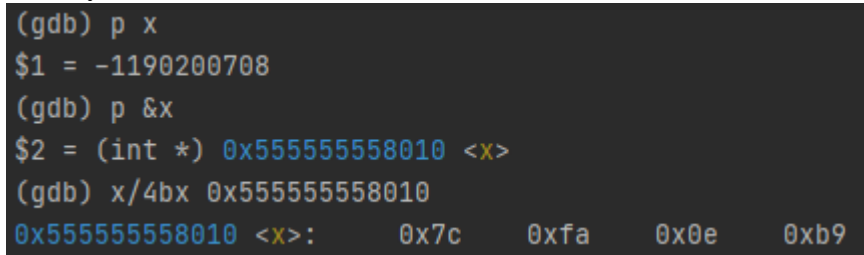
Parametric_analysis_of_main x
/mnt/d/Code/CLionProjects/C
x = -1190200708
y = 321234200204.060791
z = 1190200708-熊峰

进程已结束，退出代码为 0

```

(截图 1)

反汇编查看 x、y、z 的地址，每字节的内容：截图 2，标注说明

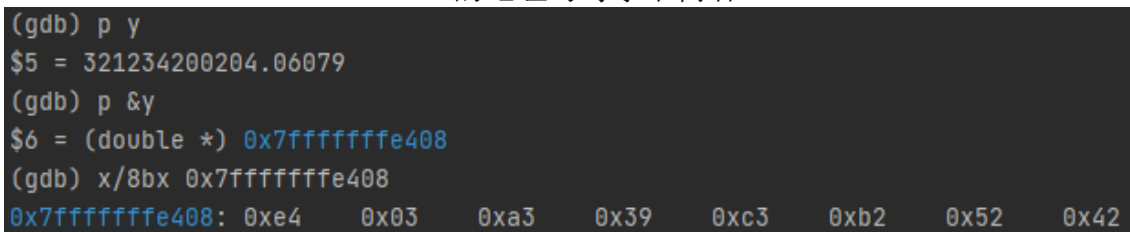


```

(gdb) p x
$1 = -1190200708
(gdb) p &x
$2 = (int *) 0x55555558010 <x>
(gdb) x/4bx 0x55555558010
0x55555558010 <x>: 0x7c 0xfa 0x0e 0xb9

```

(x 的地址与每字节内容)

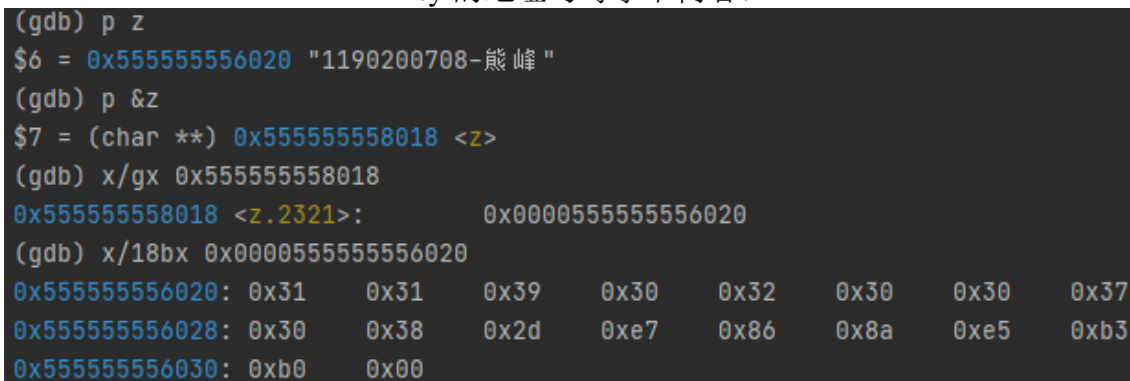


```

(gdb) p y
$5 = 321234200204.06079
(gdb) p &y
$6 = (double *) 0x7fffffffef408
(gdb) x/8bx 0x7fffffffef408
0x7fffffffef408: 0xe4 0x03 0xa3 0x39 0xc3 0xb2 0x52 0x42

```

(y 的地址与每字节内容)



```

(gdb) p z
$6 = 0x55555556020 "1190200708-熊峰"
(gdb) p &z
$7 = (char **) 0x55555558018 <z>
(gdb) x/gx 0x55555558018
0x55555558018 <z.2321>: 0x000055555556020
(gdb) x/18bx 0x000055555556020
0x55555556020: 0x31 0x31 0x39 0x30 0x32 0x30 0x30 0x37
0x55555556028: 0x30 0x38 0x2d 0xe7 0x86 0x8a 0xe5 0xb3
0x55555556030: 0xb0 0x00

```

(z 的地址与每字节内容)

反汇编查看 x、y、z 在代码段的表示形式。截图 3，标注说明

Disassembly of section .data:

0000000000004000 <\_\_data\_start>:

...

0000000000004008 <\_\_dso\_handle>:

4008: 08 40 00 00 00 00 00 00 .@.....

0000000000004010 <x>:

4010: 7c fa 0e b9 00 00 00 00 |.....

(x 的表现形式)

0000000000002000 <\_IO\_stdin\_used>:

```

2000: 01 00          add    %eax, (%rax)
2002: 02 00          add    (%rax), %al
2004: 00 00          add    %al, (%rax)
2006: 00 00          add    %al, (%rax)
2008: 25 64 0a 00 25 and    $0x25000a64, %eax
200d: 66 0a 00      data16 or    (%rax), %al
2010: 31 31          xor    %esi, (%rcx)
2012: 39 30          cmp    %esi, (%rax)
2014: 32 30          xor    (%rax), %dh
2016: 30 37          xor    %dh, (%rdi)
2018: 30 38          xor    %bh, (%rax)
201a: 2d e7 86 8a e5 sub    $0xe58a86e7, %eax
201f: b3 b0          mov    $0xb0, %bl
2021: 00 00          add    %al, (%rax)
2023: 00 00          add    %al, (%rax)
2025: 00 00          add    %al, (%rax)
2027: 00 e4          add    %ah, %ah
2029: 03 a3 39 c3 b2 52 add    0x52b2c339(%rbx), %esp
202f: 42            rex.X

```

(y 的表现形式)

xyz: file format elf64-x86-64

Contents of section .rodata:

```

2000 01000200 00000000 25640a00 25660a00 .....%d..%f..
2010 31313930 32303037 30382de7 868ae5b3 1190200708-.....
2020 b0000000 00000000 e403a339 c3b25242 .....9..RB

```

Disassembly of section .rodata:

0000000000002000 <\_IO\_stdin\_used>:

```

2000: 01 00 02 00 00 00 00 00 25 64 0a 00 25 66 0a 00 .....%d..%f..
2010: 31 31 39 30 32 30 30 37 30 38 2d e7 86 8a e5 b3 1190200708-.....
2020: b0 00 00 00 00 00 00 00 e4 03 a3 39 c3 b2 52 42 .....9..RB

```

(z 的表现形式)

x 与 y 在 编译 阶段转换成补码与 ieee754 编码。

数值型常量与变量在存储空间上的区别是：数值常量一般存放在.rodata 段，初始化的静态变量和初始化的全局变量储存在.data 段，未初始化的全局变量存放在.bss 段，局部变量储存在栈，动态分配变量存储在堆。

字符串常量与变量在存储空间上的区别是：未初始化的字符串常量存在.bss 段，初始化的字符串常量存放在.rodata 段；字符串变量存储在.data 段中。

常量表达式在计算机中处理方法是：由于乘除法的时间惩罚过高，在编译期间，编译器会优先使用左移右移指令代替乘除法，并将所得结果保存。

### 3.3 main 的参数分析 (2 分)

反汇编查看 x、y、z 的地址，argc 的地址，argv 的地址与内容，截图 4

```
(gdb) p argv[1]
$1 = 0x7fffffffedf9 "-1190200708"
(gdb) p argv[2]
$2 = 0x7fffffffef05 "321234200204.060810"
(gdb) p argv[3]
$3 = 0x7fffffffef19 "1190200708-熊峰"
```

(x、y、z 的地址)

```
(gdb) p &argc
$4 = (int *) 0x7fffffffefac
```

(argc 地址)

```
(gdb) p &argv
$12 = (char **) 0x7fffffffefaa0
(gdb) p argv
$13 = (char **) 0x7fffffffefba8
(gdb) x/gx 0x7fffffffefba8
0x7fffffffefba8: 0x00007fffffffeda6
```

(argv 的地址)

```
(gdb) x/133bx 0x00007fffffffeda6
0x7fffffffeda6: 0x2f  0x6d  0x6e  0x74  0x2f  0x64  0x2f  0x43
0x7fffffffedae: 0x6f  0x64  0x65  0x2f  0x43  0x4c  0x69  0x6f
0x7fffffffedb6: 0x6e  0x50  0x72  0x6f  0x6a  0x65  0x63  0x74
0x7fffffffedbe: 0x73  0x2f  0x43  0x53  0x41  0x50  0x50  0x5f
0x7fffffffedc6: 0x4c  0x61  0x62  0x32  0x2f  0x63  0x6d  0x61
0x7fffffffedce: 0x6b  0x65  0x2d  0x62  0x75  0x69  0x6c  0x64
0x7fffffffedd6: 0x2d  0x64  0x65  0x62  0x75  0x67  0x2f  0x50
0x7fffffffedde: 0x61  0x72  0x61  0x6d  0x65  0x74  0x72  0x69
0x7fffffffede6: 0x63  0x5f  0x61  0x6e  0x61  0x6c  0x79  0x73
0x7fffffffedee: 0x69  0x73  0x5f  0x6f  0x66  0x5f  0x6d  0x61
0x7fffffffedf6: 0x69  0x6e  0x00  0x2d  0x31  0x31  0x39  0x30
0x7fffffffedfe: 0x32  0x30  0x30  0x37  0x30  0x38  0x00  0x33
0x7fffffffee06: 0x32  0x31  0x32  0x33  0x34  0x32  0x30  0x30
0x7fffffffee0e: 0x32  0x30  0x34  0x2e  0x30  0x36  0x30  0x38
0x7fffffffee16: 0x31  0x30  0x00  0x31  0x31  0x39  0x30  0x32
0x7fffffffee1e: 0x30  0x30  0x37  0x30  0x38  0x2d  0xe7  0x86
```

(argv 的内容)

### 3.4 指针与字符串的区别 (2 分)

cstr 的地址与内容截图，pstr 的内容与截图，截图 5  
cstr 的地址与内容：

```
(gdb) p &cstr
$3 = (char (*)[17]) 0x7fffffff400
(gdb) p cstr
$4 = "1190200708熊峰"
```

pstr 的地址与内容：

```
(gdb) p &pstr
$5 = (char **) 0x7fffffff3f8
(gdb) p pstr
$6 = 0x55555556004 "1190200708熊峰"
```

pstr 修改内容会出现什么问题 无法修改 pstr 内容，pstr 仅声明为指向字符串的指针，它是一个指针，而不是字符数组，编译器报错。

```
pstr = 1190200708熊峰
```

```
进程已结束，退出代码为 139
```

## 第 4 章 深入分析 UTF-8 编码

### 4.1 提交 utf8len.c 子程序

源代码包含在附件中。

```
#include <stdio.h>
int utf8len(char* string);
int main()
{
    char* string = "1190200708-熊峰";
    int len = utf8len(string);
    if (len == -1)
        printf("存在非法字符!\n");
    else
        printf("字符个数位:%d", len);
    return 0;
}
int utf8len(char* string)
{
    int len = 0;
    char* ptr = string;
    while (*ptr != '\0')
    {
        if ((*ptr & 0xf8) == 0xf0)
        {
            if ((*++ptr & 0xc0) == 0x80 && (*++ptr & 0xc0) == 0x80 &&
                (*++ptr & 0xc0) == 0x80)
                len++;
            else
                break;
        }
        else if ((*ptr & 0xf0) == 0xe0)
        {
            if ((*++ptr & 0xc0) == 0x80 && (*++ptr & 0xc0) == 0x80)
                len++;
            else
                break;
        }
        else if ((*ptr & 0xe0) == 0xc0)
        {
            if ((*++ptr & 0xc0) == 0x80)
                len++;
            else
                break;
        }
        else if ((*ptr & 0x80) == 0x00)
            len++;
        ptr++;
    }
}
```

```

if (*ptr != '\0')
    return -1;
else
    return len;
}

```

## 4.2 C 语言的 strcmp 函数分析

分析论述：strcmp 到底按照什么顺序对汉字排序

```

#include <stdio.h>
#include <string.h>
int main()
{
    char str1[30] = "啊";
    char str2[30] = "吧";
    char str3[30] = "陈";
    printf( format: " %d\n", strcmp(str1,str2));
    printf( format: " %d\n", strcmp(str1,str3));
    printf( format: " %d\n", strcmp(str2,str3));
    return 0;
}

```

/mnt/d/Code/CLionProject  
5  
-4  
-4  
进程已结束，退出代码为 0

分析：由 `strcmp(str1,str2)=5` 及 `strcmp(str2,str3)=-4` 可知，strcmp 与汉字的字典序无关，且与姓氏笔画数无关。

实验环境为 Linux，采用 utf-8 编码，三个字符串第一个汉字所对应编码分别为 0xe5958a,0xe590a7,0xe99988,满足 strcmp 的结果。

故 strcmp 应按照汉字对应编码排序。

## 4.3 讨论：按照姓氏笔画排序的方法实现

分析论述：应该怎么实现呢？

首先分析笔画排序的规则：

- 一、笔画数由少到多的原则。
- 二、笔画数相同的，按姓氏起次笔排序的原则。
- 三、同姓一般以姓名的第二个字的笔画多少为序。
- 四、姓氏的笔画数相同、起次笔顺序一致的，按姓氏的字形结构排序的原则。
- 五、对于姓氏的笔画数相同、起次笔顺序一致，且字形结构相同的，左右形汉字的排序要遵循——按“左偏旁”笔画数由少到多的顺序排定之原则。

根据以上规则，可以对汉字所对应编码构建索引表，并根据以上规则，赋具体的值，在此基础上，构造比较器，实现以较低算法复杂度，实现查找比较。

## 第 5 章 数据变换与输入输出

### 5.1 提交 `cs_atoi.c`

源代码包含在附件中。

### 5.2 提交 `cs_atof.c`

源代码包含在附件中。

### 5.3 提交 `cs_itoa.c`

源代码包含在附件中。

### 5.4 提交 `cs_ftoa.c`

源代码包含在附件中。

### 5.5 讨论分析 OS 的函数对输入输出的数据有类型要求吗

论述如下：

根据 `<unistd.h>` 中的定义：

```
extern ssize_t read (int __fd, void *__buf, size_t __nbytes) __wur;

/* Write N bytes of BUF to FD. Return the number written, or -1.

   This function is a cancellation point and therefore not marked with
   __THROW. */
extern ssize_t write (int __fd, const void *__buf, size_t __n) __wur;
```

OS 的函数分别通过调用 `read` 和 `write` 函数来执行输入和输出。

`read` 函数分析：

`__nbytes` 是请求读取的字节数，读入的数据保存在缓冲区 `__buf` 中，文件当前读写位置向后移。返回值类型是 `ssize_t`，表示有符号的 `size_t`，这样既可以返回正的字节数、0（表示到达文件末尾）也可以返回负值-1（表示出错）。`read` 函数返回

时，返回值说明了\_\_buf 中前多少个字节是刚读上来的。有些情况下，实际读到的字节数（返回值）会小于请求读的字节数\_\_nbytes。

write 函数分析：

返回值类型是 ssize\_t，表示有符号的 size\_t，成功返回写入的字节数，出错返回-1，写常规文件时，write 的返回值通常等于请求写的字节数。



## 第 6 章 整数表示与运算

### 6.1 提交 fib\_dg.c

源代码包含在附件中。

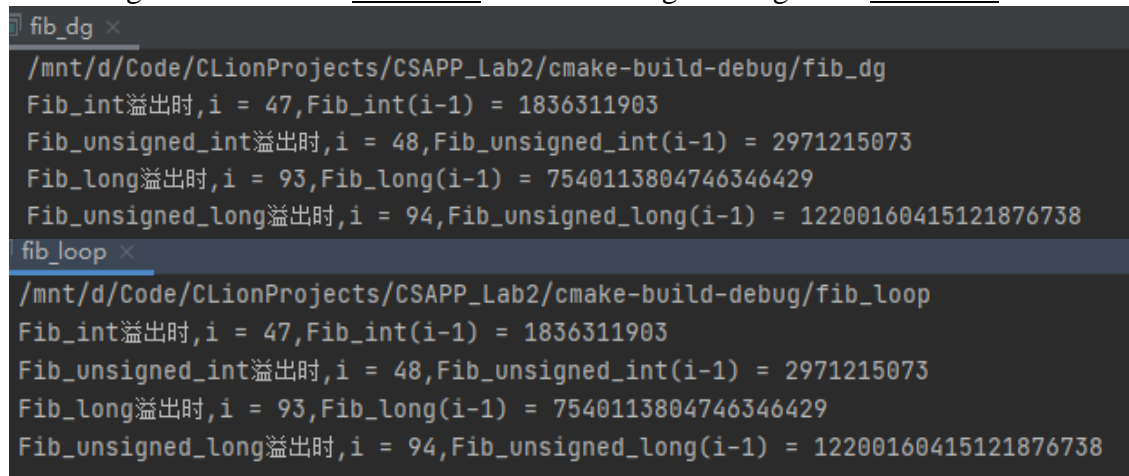
### 6.2 提交 fib\_loop.c

源代码包含在附件中。

### 6.3 fib 溢出验证

int 时从 n= 47 时溢出, long 时 n= 93 时溢出。

unsigned int 时从 n= 48 时溢出, unsigned long 时 n= 94 时溢出。



```
fib_dg x
/mnt/d/Code/CLionProjects/CSAPP_Lab2/cmake-build-debug/fib_dg
Fib_int溢出时,i = 47,Fib_int(i-1) = 1836311903
Fib_unsigned_int溢出时,i = 48,Fib_unsigned_int(i-1) = 2971215073
Fib_long溢出时,i = 93,Fib_long(i-1) = 7540113804746346429
Fib_unsigned_long溢出时,i = 94,Fib_unsigned_long(i-1) = 12200160415121876738

fib_loop x
/mnt/d/Code/CLionProjects/CSAPP_Lab2/cmake-build-debug/fib_loop
Fib_int溢出时,i = 47,Fib_int(i-1) = 1836311903
Fib_unsigned_int溢出时,i = 48,Fib_unsigned_int(i-1) = 2971215073
Fib_long溢出时,i = 93,Fib_long(i-1) = 7540113804746346429
Fib_unsigned_long溢出时,i = 94,Fib_unsigned_long(i-1) = 12200160415121876738
```

### 6.4 除以 0 验证:

除以 0: 截图 1

除以极小浮点数, 截图:

被除数类型为 int 时, int/0, 编译器报错。

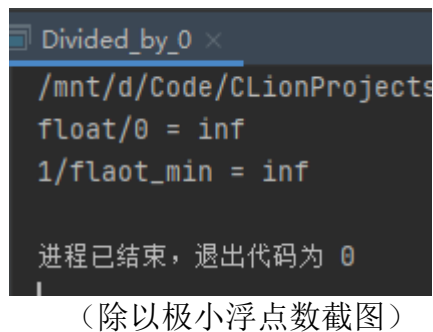
被除数类型为 float 时, float/0 = inf。



```

Divided_by_0 x
/mnt/d/Code/CLionProjects float/0 = inf
进程已结束，退出代码为 136 (被除数为 int 时)
Divided_by_0 x
/mnt/d/Code/CLionProjects float/0 = inf
进程已结束，退出代码为 0 (被除数为 float 时)

```



```

Divided_by_0 x
/mnt/d/Code/CLionProjects
float/0 = inf
1/flaot_min = inf
进程已结束，退出代码为 0

```

(除以极小浮点数截图)

```

#include <stdio.h>
union {
    float x;
    int y;
}float_min;
int main()
{
    float_min.y=0b00000000000000000000000000000001;
    float a=1;
    printf(format: "float/0 = %f\n",a/0);
    printf(format: "1/flaot_min = %f\n",a/float_min.x);
    return 0;
}

```

## 第 7 章 浮点数据的表示与运算

### 7.1 正数表示范围

写出 float/double 类型最小的正数、最大的正数（非无穷）

float:

最小的正数:  $2^{(-149)}$

最大的正数:  $2^{(128)}-2^{(104)}$

double:

最小的正数:  $2^{(-1074)}$

最大的正数:  $2^{(1024)}-2^{(971)}$

### 7.2 浮点数的编码计算

(1) 按步骤写出 float 数 -1.1 的浮点编码计算过程，写出该编码在内存中从低地址字节到高地址字节的 16 进制数值

将 1.1 转化为二进制，1.0001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 101...

由于 -1.1 为负数，故符号位  $s = 1$ 。

其整数部分为 1，故阶码  $\text{exp} - \text{bias} = 0$ ，由于  $\text{bias} = 127$ ，故  $\text{exp} = 127$ ，则其二进制表示为 01111111，即  $\text{exp} = 01111111$ 。

小数部分为 0001 1001 1001 1001 1001 1001 1... 根据舍入原则，应对小数部分进行偶舍入， $\text{frac} = 0001 1001 1001 1001 1001 101$ 。

综上：-1.1 的浮点编码为 1011 1111 1000 1100 1100 1100 1100 1101，其转化为 16 进制数值为 bf8c cccd。由于机器为小端法机器，因此该编码在内存中，从低地址到高地址的 16 进制数应为 cd cc 8c bf。

(2) 验证：编写程序，输出值为 -1.1 的浮点变量其各内存单元的数值，截图。

```
float_code x
/mnt/d/Code/CLionProjects/CSAPP_Lab2/cmake-bui
float num = -1.100000:
16进制编码为: bf8cccd
各内存单元数值为: cd cc 8c bf
进程已结束，退出代码为 0
```

```
(gdb) p float_code
$3 = {x = -1081291571, y = -1.10000002}
(gdb) p &float_code
$4 = (union {...} *) 0x555555558014 <float_code>
(gdb) x/4bx 0x555555558014
0x555555558014 <float_code>:    0xcd    0xcc    0x8c    0xbf
```

### 7.3 特殊浮点数值编码

(1) 构造多 float 变量，分别存储+0-0，最小浮点正数，最大浮点正数、最小正的规格化浮点数、正无穷大、Nan,并打印最可能的精确结果输出（十进制/16 进制）。截图。

```
floatx x
/mnt/d/Code/CLionProjects/CSAPP_Lab2/cmake-build-debug/floatx
+0 = 0.000000 , 编码 = 00000000 , 每字节为 00 00 00 00
-0 = -0.000000 , 编码 = 80000000 , 每字节为 00 00 00 80
+min = 0.000000 , 编码 = 00000001 , 每字节为 01 00 00 00
+max = 340282346638528859811704183484516925440.000000 , 编码 = 7f7fffff , 每字节为 ff ff 7f 7f
+min_normalized_number = 0.000000 , 编码 = 00800000 , 每字节为 00 00 80 00
infinity = inf , 编码 = 7f800000 , 每字节为 00 00 80 7f
nan_float = nan , 编码 = 7f800001 , 每字节为 01 00 80 7f
进程已结束，退出代码为 0
```

- (2) 提交子程序 floatx.c  
源代码包含在附件中。

### 7.4 浮点数除 0

- (1) 编写 C 程序，验证 C 语言中 float 除以 0/极小浮点数后果，截图

```
float0 x
/mnt/d/Code/CLionProjects/CSAPP_La
float / 0 = inf
float / min_float = inf
进程已结束，退出代码为 0
```

- (2) 提交子程序 float0.c  
源代码包含在附件中

### 7.5 Float 的微观与宏观世界

按照阶码区域写出 float 的最大密度区域范围及其密度，最小密度区域及其密度(区域长度/表示的浮点个数):  $[2^{(-149)}, 2^{-2^{(-149)}}]$  、  $2^{(24)}/(2-2^{(-148)})$  、

$$(2^{127}, 2^{128}-2^{104}) \quad , \quad (2^{23}-2)/(2^{127}-2^{104})$$

简要分析:

最大密度区:

对 float 的 IEEE754 编码分析, 当阶码为 0000 0000 时与阶码为 0000 0001 时, 由于此时乘以的 2 的阶数相同, float 类型数的密度相同。

故 float 的密度最大区间为非规格化数部分+阶码为 0000 0001 的规格化数部分。

在正半轴或负半轴上考虑, 有限区间内, 由于 0 的存在, float 的数量加一, 大于考虑整个实数轴时的密度, 故此时为密度最大。

由上, 仅考虑正半轴, float 的编码范围为 0 0000 0000 0000 0000 0000 0000 000 至 0 0000 0001 1111 1111 1111 1111 111 111, 即  $[2^{-(149)}, 2-2^{-(149)}]$ 。

密度最大区域的数的个数为  $2^{24}$ , 其密度为  $2^{(24)}/(2-2^{-(148)})$

最小密度区:

对 float 的 IEEE754 编码分析, 当阶码为 1111 1110 时, 此时 2 的阶数最大, 此时 float 的密度最小。

float 的编码范围为 0 1111 1110 0000 0000 0000 0000 0000 000 至 0 1111 1110 1111 1111 1111 1111 1111 111, 即  $(2^{127}, 2^{128}-2^{104})$ , 此时在此区间的数的个数为  $2^{23}-2$ , 则此时密度为  $(2^{23}-2)/(2^{127}-2^{104})$

最小正数变成十进制科学记数法, 最可能能精确到多少 1.4012984e-45  
最大正数变成十进制科学记数法, 最可能能精确到多少 1.701411733192e+38

简要分析:

最小正数变成十进制科学记数法:

最小 float 型数的编码为 0 0000 0000 0000 0000 0000 0000 0000 001, 此时, float 的数值为  $2^{(-23)} * 2^{(-126)} = 2^{(-149)}$ . 转化为 10 进制数为 1.4012984e-45.

最大正数变成十进制科学记数法:

最大 float 型数的编码为 0 1111 1110 1111 1111 1111 1111 1111 111, 此时, float 的数值为  $2^{127}-2^{103}$ , 转化为 10 进制数为 1.70141173319265e+38.

## 7.6 讨论: 任意两个浮点数的大小比较

论述比较方法以及原因。

比较方法:

应设置一个较小的浮点数, 作为一个精度常量, 若两个浮点数相减结果的绝对值小于这个精度常量, 则在一定限度内, 这两个浮点数是相等的, 反之则不相等。

原因:

浮点数在计算机中并不是精确保存的, 若仅仅简单的用  $=$  运算符, 是有失偏颇的, 在某些相等的情况下, 可能会引发错误的结果。由于其常常伴随一定误差, 故若相减的结果足够小, 可以近似认为两个浮点数相等。

## 第 8 章 舍位平衡的讨论

### 8.1 描述可能出现的问题

在数据统计应用中,常常会根据精度呈现或者单位换算的要求对数据四舍五入的操作,然而这种处理方式可能会带来一定的隐患,由于数据的单位改变,常常会出现数据不平衡的现象。

例如: 13,451.00 元 + 45,150.00 元 + 2,250.00 元 - 5,649.00 元 = 55,202.00 元  
若单位变成万元,仍然保留两位小数,根据 4 舍 5 入的原则: 1.35 万元 + 4.52 万元  
+ 0.23 万元 - 0.56 万元 = 5.54 万元,出现 0.02 万的误差,平衡被打破。

### 8.2 给出完美的解决方案

在数据统计时,显然在舍位前后会累计误差,导致数据不平衡,为了保证舍位后的平衡的关系,应分别改变各个原始数据舍位后的结果。完成舍位平衡操作,就是在完成舍位操作时,消除舍位产生的误差。

- (1) 设所有数据为 A, 则 A 的和为 A.sum(), 并对其求整, round(A.sum())
- (2) 计算对 A 求整后的数据 B, 计算 B 的和 B.sum()
- (3) 计算 round(B.sum()), 即对 B.sum()求整
- (4) 计算  $a = \text{round}(\text{A.sum}()) - \text{B.sum}()$
- (5) 对 A 中的所有数据的绝对值排序
- (6) 计算 a 的绝对值, 并根据 a 的绝对值, 调整数据。

## 第 9 章 总结

### 9.1 请总结本次实验的收获

- (1)学会并掌握使用 gdb 调试程序的指令;
- (2)学会并掌握使用 objdump 反汇编;
- (3)学会并掌握使用 readelf 查看 linux 下的可执行文件;
- (4)了解了汉字的 utf-8 编码与 unicode 编码;
- (5)了解了 main 函数的参数;
- (6)熟悉了 float 的 IEEE754 编码;
- (7)熟悉链接的步骤。

### 9.2 请给出对本次实验内容的建议

- (1)实验中存在部分超过当前学习的内容,理解起来稍有困难。
- (2)实验中遇到部分无法理解的情况,例如:反汇编查看 xyz 的地址。
- (3)实验中存在部分重复情况,例如:除以 0 验证的内容重复出现。
- (4)希望实验 PPT 等内容可以更加翔实,存在部分阅读题目后无法理解情况。

注:本章为酌情加分项。

## 参考文献

- [1] RANDELE.BRYANT, DAVIDR.O 'HALLARON. 深入理解计算机系统[M]. 机械工业出版社, 2011.