

哈尔滨工业大学

实验报告

实 验（二）

题 目 DataLab 数据表示

专 业 计算机类

学 号 1190201215

班 级 1903007

学 生 冯开来

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

第 1 章 实验基本信息	- 4 -
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 -
第 2 章 实验环境建立	- 6 -
2.1 UBUNTU 下 CODEBLOCKS 安装	- 6 -
2.2 64 位 UBUNTU 下 32 位运行环境建立	- 7 -
第 3 章 C 语言的数据类型与存储	- 8 -
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 -
第 4 章 深入分析 UTF-8 编码	- 14 -
4.1 提交 UTF8LEN.C 子程序	- 14 -
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 -
第 5 章 数据变换与输入输出	- 14 -
5.1 提交 CS_ATOI.C	- 14 -
5.2 提交 CS_ATOF.C	- 14 -
5.3 提交 CS_ITOA.C	- 14 -
5.4 提交 CS_FTOA.C	- 14 -
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 -
第 6 章 整数表示与运算	- 15 -
6.1 提交 FIB_DG.C	- 15 -
6.2 提交 FIB_LOOP.C	- 15 -
6.3 FIB 溢出验证	- 15 -
6.4 除以 0 验证:	- 16 -
第 7 章 浮点数据的表示与运算	- 17 -
7.1 正数表示范围	- 17 -
7.2 浮点数的编码计算	- 17 -
7.3 特殊浮点数值的编码	- 18 -
7.4 浮点数除 0	- 18 -

7.5 FLOAT 的微观与宏观世界	- 18 -
7.6 讨论：任意两个浮点数的大小比较.....	- 18 -
第 8 章 舍位平衡的讨论	- 20 -
8.1 描述可能出现的问题.....	- 20 -
8.2 给出完美的解决方案.....	- 20 -
第 9 章 总结	- 22 -
9.1 请总结本次实验的收获.....	- 22 -
9.2 请给出对本次实验内容的建议.....	- 22 -
参考文献.....	- 23 -

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

X64 CPU; 2GHz; 2G RAM; 256GHD Disk 以上

1.2.2 软件环境

Windows7 64 位以上; VirtualBox/Vmware 11 以上; Ubuntu 16.04 LTS 64 位

1.2.3 开发工具

Visual Studio 2010 64 位以上; CodeBlocks; vi/vim/gpedit+gcc

1.3 实验预习

- 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

1.递归方式:

$f(n) = f(n-1) + f(n-2)$, $n \geq 3$;

$f(1) = f(2) = 1$;

其时间复杂度: $O(2^n)$

2.循环方式:

```
for(int i = 3; i < n; i++)  
{  
    temp = res;  
    res = pre + res;  
    pre = temp;  
}
```

其时间复杂度: $O(n)$

int 为 47

long int 为 47

unsigned int 为 48

unsigned long 为 48

- 写出 float/double 类型最小的正数、最大的正数（非无穷）

float 最小值: 1.4E-45

最大值: 3.4028235E38

double 最小值: 4.9E-324

最大值: 1.7976931348623157E308

- 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

cd cc 8c bf

- 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）

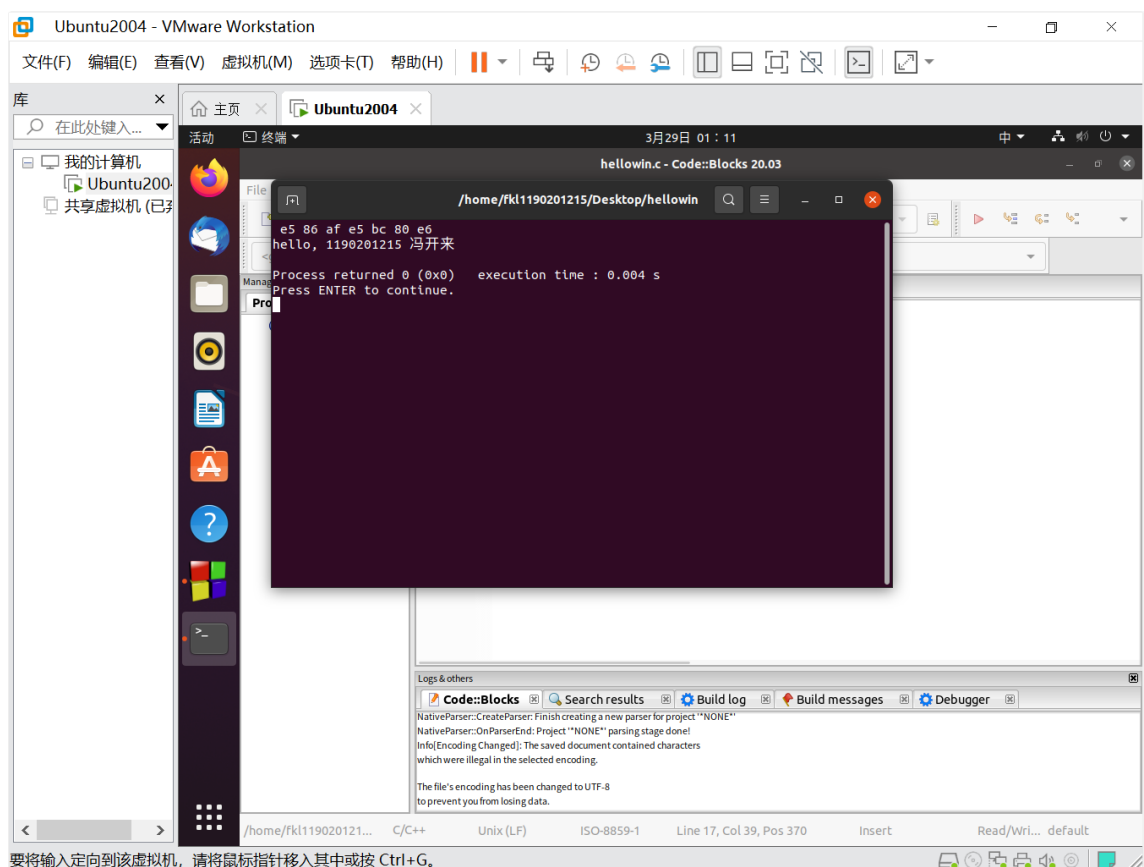
最大: $-1.1111(23 \text{ 个 } 1) \times 2^{-126} \sim 1.1111(23 \text{ 个 } 1) \times 2^{-126}$ (包含正负浮点数);
 2^{25} 个浮点数 ;

最小: $1.0000(23 \text{ 个 } 0) \times 2^{127} \sim 1.1111(23 \text{ 个 } 1) \times 2^{127}$;
 2^{23} 个浮点数

第 2 章 实验环境建立

2.1 Ubuntu 下 CodeBlocks 安装

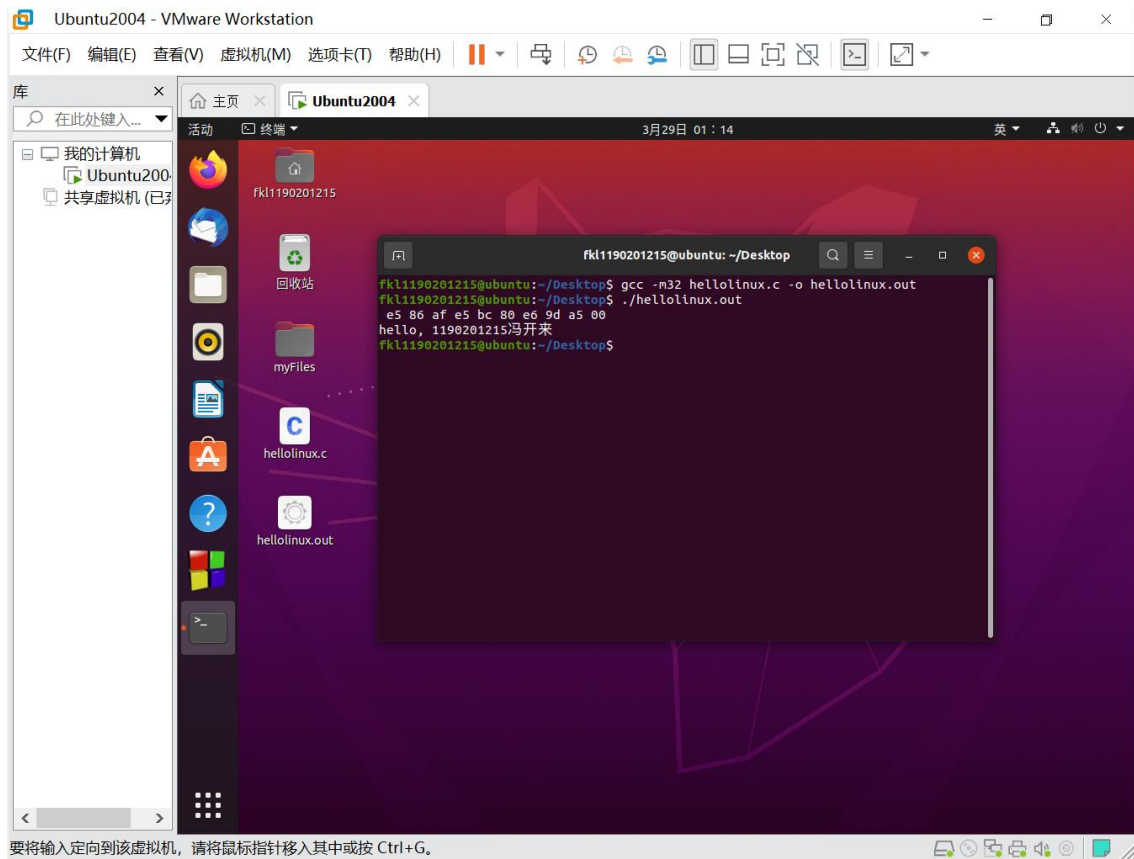
CodeBlocks 运行界面截图：编译、运行 hellolinux.c



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)

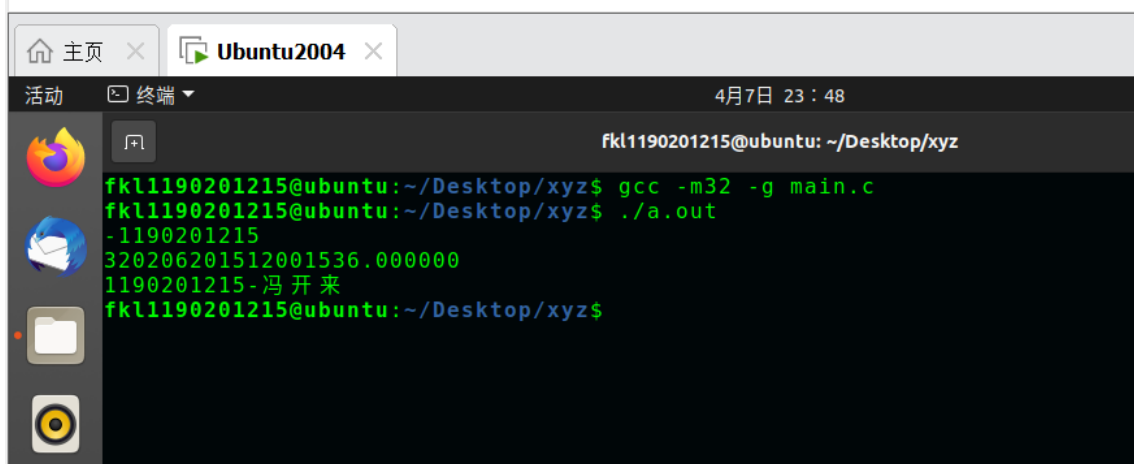
	Win/VS/x86	Win/VS/x64	Win/CB/32	Win/CB/64	Linux/CB/32	Linux/CB/64
char	1byte	1byte	1byte	1byte	1byte	1byte
short	2byte	2byte	2byte	2byte	2byte	2byte
int	4byte	4byte	4byte	4byte	4byte	4byte
long	4byte	4byte	4byte	4byte	4byte	8byte
long long	8byte	8byte	8byte	8byte	8byte	8byte
float	4byte	4byte	4byte	4byte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

C 编译器对 sizeof 的实现方式:

由编译器来计算，编译阶段就计算出结果了，在运行时就是个常量。编译阶段可以确定数据类型，根据数据类型换算数据的长度。

3.2 数据的位置-地址 (2 分)

打印 x、y、z 输出的值:



```

fkl1190201215@ubuntu: ~/Desktop/xyz
fkl1190201215@ubuntu:~/Desktop/xyz$ gcc -m32 -g main.c
fkl1190201215@ubuntu:~/Desktop/xyz$ ./a.out
-1190201215
320206201512001536.000000
1190201215-冯开来
fkl1190201215@ubuntu:~/Desktop/xyz$
  
```


反汇编查看 x、y、z 的地址，每字节的内容：

x 的地址：0x56559008；

内容：81 f8 0e b9

y 的地址：0xffffcffc；

内容：3b 33 8e 5c

z 的地址：0x5655900c；

内容：31 31 39 30 32 30 31 32 31 35 2d e5 86 af e5 bc 80 e6 9d a5 00

```

(gdb) n
-1190201215
320206201512001536.000000
1190201215-泻开来
9
(gdb) x /4xb &x
0x56559008: <x>: 0x81 0xf8 0x0e 0xb9
(gdb) x /4xb &y
0xffffcffc: 0x3b 0x33 0x8e 0x5c
(gdb) x /20b &z
0x5655900c: <z.2432>: 0x31 0x31 0x39 0x30 0x32 0x30 0x31 0x32 0x31 0x35 0x2d 0xe5 0x86 0xaf 0xe5 0xbc 0x80 0xe6 0x9d 0xa5
0x56559014: <z.2432+8>: 0x31 0x35 0x2d 0xe5 0x86 0xaf 0xe5 0xbc
0x5655901c: <z.2432+16>: 0x80 0xe6 0x9d 0xa5
(gdb)
  
```

Labels in image: x地址 (points to 0x56559008), x内容 (points to 0x81 0xf8 0x0e 0xb9), y地址 (points to 0xffffcffc), y内容 (points to 0x3b 0x33 0x8e 0x5c), z地址 (points to 0x5655900c), z内容 (points to the first row of the z array).

反汇编查看 x、y、z 在代码段的表示形式。截图 3，标注说明

x 的代码段：

cmp \$0x3131b90e, %eax

```

Disassembly of section .data:
00004000 <__data_start>:
4000: 00 00          add    %al, (%eax)
...
00004004 <__dso_handle>:
4004: 04 40          add    $0x40, %al
...
00004008 <x>:
4008: 81 f8 0e b9    cmp    $0x3131b90e, %eax
...
0000400c <z.2432>:
400c: 31 31          xor    %esi, (%ecx)
400e: 39 30          cmp    %esi, (%eax)
4010: 32 30          xor    (%eax), %dh
4012: 31 32          xor    %esi, (%edx)
4014: 31 35 2d e5 86 af  xor    %esi, 0xaf86e52d
401a: e5 bc          in     $0xbc, %eax
401c: 80 e6 9d          and    $0x9d, %dh
401f: a5            movsl  %ds:(%esi), %es:(%edi)
...
Disassembly of section .bss:
  
```

Label in image: x代码段 (points to the highlighted instruction).

y 的代码段:

```
cmp      (%ebx), %esi
.byte    0x8e
pop      %esp
```

```

4月7日 23:58
fkl1190201215@ubuntu: ~/Desktop/xyz

00002004 <_IO_stdin_used>:
2004:      01 00      add    %eax, (%eax)
2006:      02 00      add    (%eax), %al
2008:      25 64 0a 25 6c      and    $0x6c250a64, %eax
200d:      66 0a 25 73 0a 00 00      data16 or 0xa73, %ah
2014:      3b 33      cmp    (%ebx), %esi
2016:      8e          .byte  0x8e
2017:      5c          pop    %esp

Disassembly of section .eh_frame_hdr:
00002018 <__GNU_EH_FRAME_HDR>:
2018:      01 1b      add    %ebx, (%ebx)

```

z 的代码段:

```
xor      %esi, (%ecx)
cmp      %esi, (%eax)
xor      (%eax), %dh
xor      %esi, (%edx)
xor      %esi, 0xaf86e52d
in       $0xbc, %eax
and      $0x9d, %dh
movsl    %ds:(%esi), %es:(%edi)
```

```

fkl1190201215@ubuntu: ~/Desktop/xyz

Disassembly of section .data:
00004000 <__data_start>:
4000:      00 00      add    %al, (%eax)
...

00004004 <__dso_handle>:
4004:      04 40      add    $0x40, %al
...

00004008 <x>:
4008:      7f 07      jg     4011 <z.2430+0x5>
400a:      f1          icebp
400b:      46          inc    %esi

0000400c <z.2430>:
400c:      31 31      xor    %esi, (%ecx)
400e:      39 30      cmp    %esi, (%eax)
4010:      32 30      xor    (%eax), %dh
4012:      31 32      xor    %esi, (%edx)
4014:      31 35 2d e5 86 af      xor    %esi, 0xaf86e52d
401a:      e5 bc      in     $0xbc, %eax
401c:      80 e6 9d      and    $0x9d, %dh
401f:      a5          movsl  %ds:(%esi), %es:(%edi)

```

x 与 y 在__汇编__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：

数值型常量一般存放在.rodata 段、常量存储区、代码段(.text)中；

数值型变量若为未初始化的全局变量将存放在.bss 段，初始化的全局变量和初始化的静态变量将存放在.data 段，函数内的局部变量、传递的函数参数一般存放在栈上，动态分配的变量在堆中。

字符串常量与变量在存储空间上的区别是：

未初始化的全局字符串存放在 bss 段；全局的初始化的字符串常量内容存储在 rodata 段；全局初始化后的字符串变量存储在 data 段；函数内的局部变量（用户临时创建的）或常量中，字符串内容存放在 rodata 段，变量或常量指针放入栈中；静态字符串变量存放在 data 段。

常量表达式在计算机中处理方法是：

在程序编译时就将值储存在内存中，不可改变，无法直接修改其内容。或者在编译阶段尽量将左移右移代替乘除法，汇编阶段计算出常量表达式的值。

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

x 的地址：0x56559008；

内容：81 f8 0e b9

y 的地址：0xffffd5e0；

内容：3b 33 8e 5c 01

z 的地址：0x5655900c；

内容：31 31 39 30 32 30 31 32 31 35 2d e5 86 af e5 bc 80 e6 9d a5 00

argc 的地址：0xffffd020

内容：01 00 00 00

argv[0]地址：0x56559008

内容：/home/fk1190201215/Desktop/arg/bin/Debug/arg

argv[1]地址：0xffffd6b8

内容：81 f8 0e b9（-学号）

argv[2]地址：0xffffd6bc

内容：3b 33 8e 5c 01（身份证）

argv[3]地址：0xffffd6c0

内容：31 31 39 30 32 30 31 32 31 35 2d e5 86 af e5 bc 80 e6 9d a5 00
（学号-姓名）

```

main.c
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[])
5  {
6      int i = 0;
7      // int x = 1190201215;
8      // float y = 320206200012060415;
9      // char z[21] = "1190201215-冯开来";
10     // argv[0] = &x;
11     // argv[1] = &y;
12     // argv[2] = &z;
13
14     while (i<argc)
15     {
16         printf("%s\n",argv[i]);
17         i++;
18     }
19     return 0;
20 }
21

```

```

arg
/home/fk11190201215/Desktop/arg/bin/Debug/arg
Process returned 0 (0x0)   execution time : 0.004 s
Press ENTER to continue.

```

```

(gdb) n
1190201215
320206201512001536.000000
1190201215-冯开来
9
(gdb) x /4xb &argc
0xffffd020: 0x01 0x00 0x00 0x00
(gdb) x /4xb &argv
0xffffd024: 0xb4 0xd0 0xff 0xff

```

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 地址: 0x56559020

cstr 内容: 31 31 39 30 32 30 31 32 31 35 2d e5 86 af e5 bc 80 e6 9d a5

pstr 地址: 0x56557008

pstr 内容: 31 31 39 30 32 30 31 32 31 35 2d e5 86 af e5 bc 80 e6 9d a5

```

(gdb) n
1190201215-冯开来
1190201215-冯开来
14  return 0;
(gdb) x /20xb &cstr
0x56559020: 0x31 0x31 0x39 0x30 0x32 0x30 0x31 0x32
0x56559028: 0x31 0x35 0x2d 0xe5 0x86 0xaf 0xe5 0xbc
0x56559030: 0x80 0xe6 0x9d 0xa5
(gdb) x /20xb &pstr
0x56559084: 0x00 0x00 0x70 0x55 0x56 0x00 0x00 0x00
0x5655908c: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x56559094: 0x00 0x00 0x00 0x00
(gdb) x /20xb 0x56557008
0x56557008: 0x31 0x31 0x39 0x30 0x32 0x30 0x31 0x32
0x56557010: 0x31 0x35 0x2d 0xe5 0x86 0xaf 0xe5 0xbc
0x56557018: 0x80 0xe6 0x9d 0xa5
(gdb)

```

pstr 修改内容会出现什么问题——用学号+姓名初始化后，若以 **pstr = “...” /... 为身份证号/这一形式赋值，将会报错，因为原先 pstr 的值（学号+姓名）存储在常量区，这种赋值语句会意图修改常量区的值，不合法；若以 pstr = “...” /... 为身份证号/的形式赋值是合法的，但此时指针指向位置发生改变，即指针指向了常量区另一个地方，该地方存有后赋值的字符串常量。

第 4 章 深入分析 UTF-8 编码

4.1 提交 utf8len.c 子程序

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

C 语言中，汉字使用的编码体系里面是两个字节的 GB 编码（Linux 下一般为 utf-8，一般为三个字节），strcmp 是通过比较有符号数字大小来判断字符串的。若 $String\ 1 > String\ 2$, return 正数，若 $String\ 1 == String\ 2$, return 0，若 $String\ 1 < String\ 2$, return 负数

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？

计算出每个字的笔画，通过笔画数来比较。需要构造一个字符串数组，索引为笔画数，若相同笔画数以横竖撇点折（或者拼音字母）的顺序排列的。再实现一个比较器，使用循环到笔画数组中查找进行比较。笔划数相同的可按拼音字母字典顺序排序。以 GB 编码为例，依据区位码的汉字顺序，依次将每个简体汉字的笔画数信息存入一笔画数组，根据所查汉字的区码和位码获得该汉字在笔画数组的位置和该字的笔画数。

第 5 章 数据变换与输入输出

5.1 提交 cs_atoi.c

5.2 提交 cs_atof.c

5.3 提交 cs_itoa.c

5.4 提交 cs_ftoa.c

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：

OS 的函数将输入输出的数据都看成字符串来处理。通过输入输出流来实现，例如在 Linux 下 `stdin` 为标准输入流，标准的输入设备默认键盘；`stdout` 为标准输出流，标准的输出设备默认屏幕；`stderr` 为标准错误流，只有程序出错时才会执行的流程。OS 先接收输入的字符串，再进行进一步处理；对于输出，也就将要输出的数据转换成字符串输出的。

如：

1.sprintf 函数

`int sprintf(char *str, const char *format, ...)` 输出 `int` 类型，输入要求为字符串地址，以及用字符串如 `%s` 等作为 `format`

2.atoi 函数 atof 函数

`int atoi(const char *nptr); double atof(const char *nptr)`

输出 `int` 类型，输入字符串首地址 输出 `double` 类型，输入字符串首地址

3.itoa 函数

`char *itoa (int value, char *str, int base);`

返回转换后字符串首地址，输入待转换 `int` 数以及转换进制

4.ftoa 函数

`char *ftoa(float f, int *status)` 返回转换后字符串首地址，输入 `float` 和基数

第 6 章 整数表示与运算

6.1 提交 `fib_dg.c`

6.2 提交 `fib_loop.c`

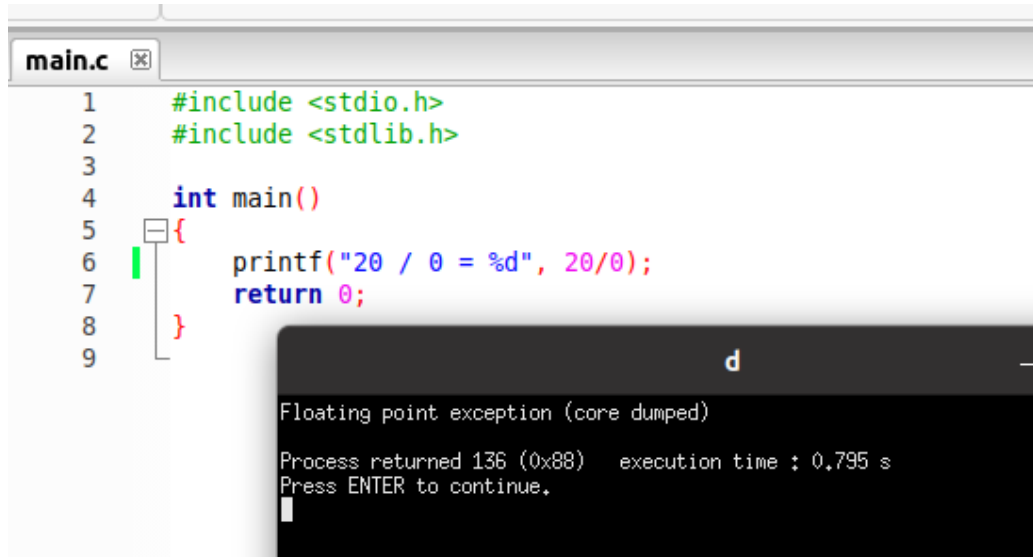
6.3 `fib` 溢出验证

`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。

6.4 除以 0 验证:

除以 0: 截图 1



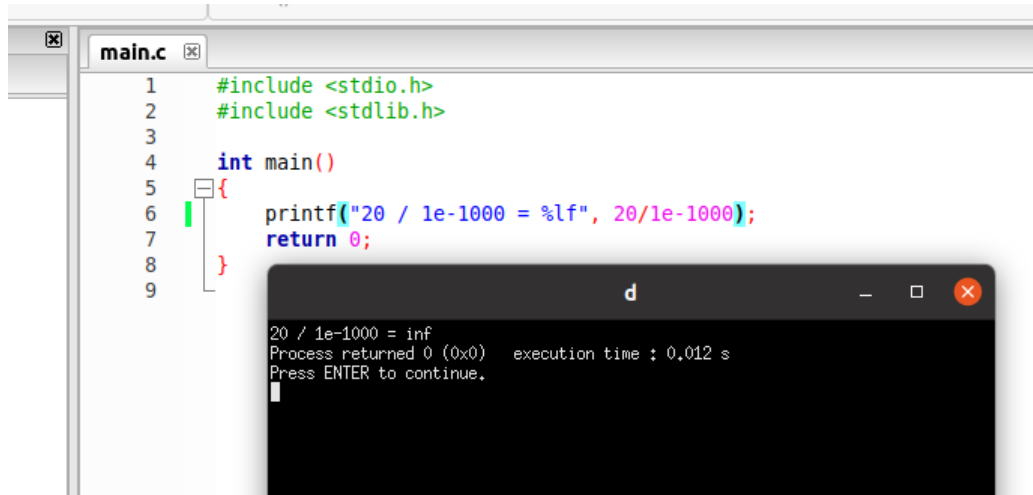
The screenshot shows a code editor with a file named `main.c` containing the following C code:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      printf("20 / 0 = %d", 20/0);
7      return 0;
8  }
```

The program is executed, and the output window (titled 'd') displays the following error message:

```
Floating point exception (core dumped)
Process returned 136 (0x88)  execution time : 0.795 s
Press ENTER to continue.
```

除以极小浮点数, 截图:



The screenshot shows a code editor with a file named `main.c` containing the following C code:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      printf("20 / 1e-1000 = %lf", 20/1e-1000);
7      return 0;
8  }
```

The program is executed, and the output window (titled 'd') displays the following result:

```
20 / 1e-1000 = inf
Process returned 0 (0x0)  execution time : 0.012 s
Press ENTER to continue.
```


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:

最小正数（十进制）： 1.401298×10^{-45}

最大正数（十进制）： $3.402823466 \times 10^{38}$

double:

最小正数（十进制）： $4.94065645841246544 \times 10^{-324}$

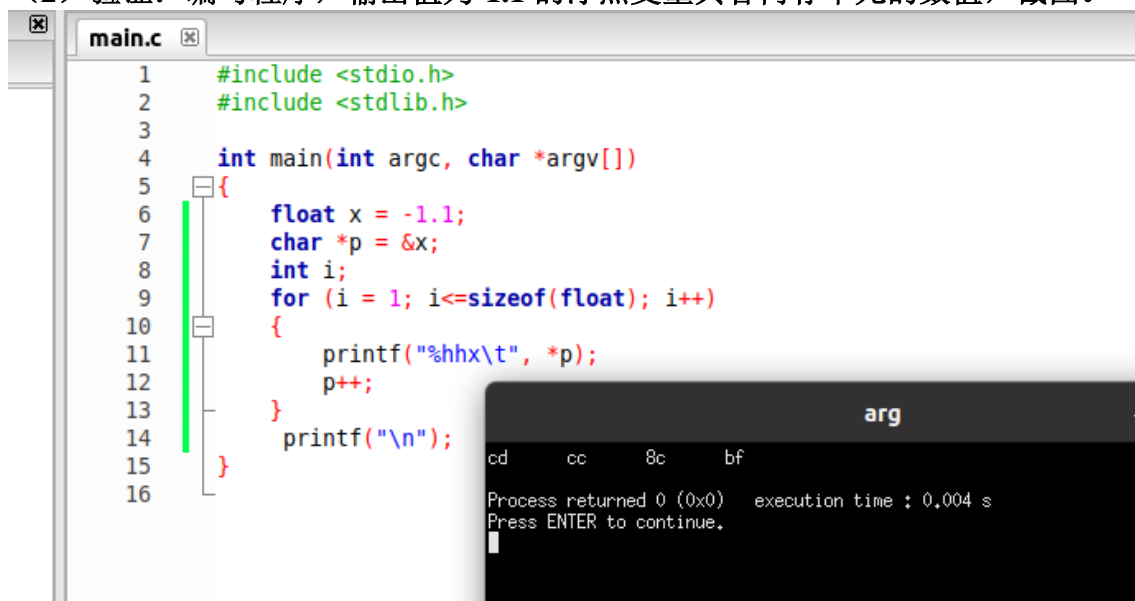
最大正数（十进制）： $1.7976931348623158 \times 10^{308}$

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

-1.1 符号位即最高位为 1，由于 $1.1 > 1$ ，故 $\text{bias} = 2^7 - 1 = 127$ ，而 $1.1 < 2$ ，故 $\text{exp}(10) = 127$ ， $\text{exp}(2) = 01111111$ ，故 $f(10) \approx 0.1$ ，将 0.1 不断乘以 2，可以得到 000[1100... (1100 循环)]，当位数达到 23 位后为 [1100...] $> 2^{-24}$ (即理论上 24 位以后为 1...1...)，向上舍入，尾数部分最后 [1100] 变为 [1101]，因此 -1.1 编码为 0xbf8cccd，从内存中低地址到高地址：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。



```
main.c
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[])
5  {
6      float x = -1.1;
7      char *p = &x;
8      int i;
9      for (i = 1; i <= sizeof(float); i++)
10     {
11         printf("%hhx\t", *p);
12         p++;
13     }
14     printf("\n");
15 }
16
```

```
arg
cd    cc    8c    bf
Process returned 0 (0x0)   execution time : 0.004 s
Press ENTER to continue.
```

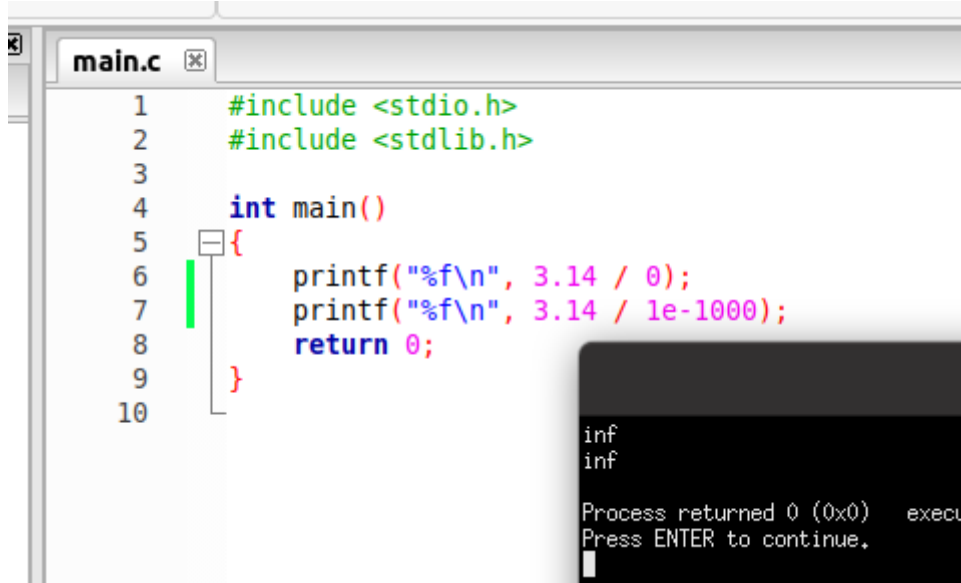
7.3 特殊浮点数值编码

(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。

(2) 提交子程序 floatx.c

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



```
main.c
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      printf("%f\n", 3.14 / 0);
7      printf("%f\n", 3.14 / 1e-1000);
8      return 0;
9  }
10
```

```
inf
inf

Process returned 0 (0x0)  execut
Press ENTER to continue.
```

(2) 提交子程序 float0.c

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）：

最大： $-1.1111(23 \text{ 个 } 1) \times 2^{-126} \sim 1.1111(23 \text{ 个 } 1) \times 2^{-126}$ (包含正负浮点数)；

2^{25} 个浮点数；

最小： $1.0000(23 \text{ 个 } 0) \times 2^{127} \sim 1.1111(23 \text{ 个 } 1) \times 2^{127}$ ；

2^{23} 个浮点数

最小正数变成十进制科学记数法，最可能精确到多少 1.401298×10^{-45}

最大正数变成十进制科学记数法，最可能精确到多少

$4.940656458412465 \times 10^{-45}$

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。

IEEE 规则中，先比较符号位正数 $>$ 负数，再比较阶码位，最后比较尾数。

编程中浮点数从 **MSB** 到 **LSB** 表示，若相差较大可直接比较大小，当两个数比较接近时，我们常常采用二分法的思想，定义精度如 $1e-7$ 等，当二者之差小于这个很小的数时，就认为二者是相等的了。为了方便起见，我们对浮点数进行比较时，一般均采用上述方法，而不直接用 $==$ 或 $!=$ 比较，因为有些数可能无法用浮点数精确表示，表示时发生舍入而影响比较结果。

第 8 章 舍位平衡的讨论

8.1 描述可能出现的问题

在数据统计中，常常会根据精度呈现或者单位换算等要求，需要对数据执行四舍五入的操作，这种操作称为舍位处理。简单直接的舍位处理有可能会带来隐患，原本平衡的数据关系可能会被打破。

例如，保留一位小数的原始的数据是 $4.5+4.5=9.0$ ，而四舍五入只保留整数部分后，平衡关系就变为 $5+5=9$ 了，看上去明显是荒谬的。

例如， $13,451.00 \text{ 元} + 45,150.00 \text{ 元} + 2,250.00 \text{ 元} - 5,649.00 \text{ 元} = 55,202.00 \text{ 元}$
现在单位变成单位万元，仍然保留两位小数，根据 4 舍 5 入的原则： $1.35 \text{ 万元} + 4.52 \text{ 万元} + 0.23 \text{ 万元} - 0.56 \text{ 万元} = 5.54 \text{ 万元}$ ，出现 0.02 万的误差，平衡被打破。

8.2 给出完美的解决方案

舍位后总计产生的误差，称为“平衡差”（真实数值-平衡后理论数值），舍位平衡其实就是消除平衡差的过程。

（1）单向舍位平衡，即如果在数据统计时，每个数据只用于一次合计，那么在处理舍位平衡时，只需要根据合计值的误差，调整使用的各项数据就可以了，这属于比较简单的情況。

考虑方法 1：我们将平衡差直接加到第一个数据，可以解决舍位平衡。但这种方法很可能会使改动后的该数据变动较大，从而影响后续操作，故该方法 pass。

考虑方法 2：我们将正精度单位（例如保留到 0.1/1，正精度单位即为 0.1/1）（如果平衡差 >0 ）或负精度单位（例如保留到-0.1/-1，正精度单位即为-0.1/-1）（如果平衡差 <0 ）加到绝对值最大的数据中，这样使得每个数据没有更改或偏差不是很大，但是寻找绝对值最大的多个数据需要进行排序算法，对于大量数据不是很适用，所以该方法 pass。

考虑方法 3：我们将所有的非零数按顺序加上正或负精度单位。考虑到在四舍五入时，0 并不会产生误差，而且如果修改数据中的 0，这样的变动会比较明显，因此在调整时将保留原始数据中的 0 不变。这样调整平衡的结果比较合理。同时这种方案避免了排序操作，效率较高，因此这种舍位平衡的规则最为常用。

（2）双向舍位平衡，即如果数据在行向和列向两个方向同时需要计算合计值，同时还需要计算所有数据的总计值。

有一些平衡差只与合计值相关，这样的平衡差称为合计平衡差。在双向舍位平衡表中，只存在一横一纵两个合计平衡差。其它的平衡差都会和具体数据相关，如 Feb 这个月最下方的平衡差，这种平衡差称为非合计平衡差。

考虑情况 1：横向与纵向的非合计平衡差符号相同。只需要调整交叉点处的数据，根据平衡差符号加减最小调整值。

考虑情况 2：同向的 2 个非合计平衡差符号相反。任选另一方向平衡差为 0 的数据，将这两个方向的数分别根据按平衡差的符号加减最小调整值。

考虑情况 3：某个合计平衡差与另一方向的非合计平衡差符号相反。调整交叉点处

的合计数据，根据合计平衡差的符号加减最小调整值。

考虑情况 4：某个合计平衡差与同方向的非合计平衡差符号相同。任选一另一方向平衡差为 0 的数据，同时调整这 2 个方向的数据

考虑情况 5：两个方向合计平衡差的符号相同。此时，只有合计数据影响了结果的平衡。任选一个非合计值，根据合计平衡差的符号加减最小调整值，同样调整这个数据的横向和纵向合计值。

综上：

- 1.对于其它的情况，说明计算有误，是无法通过 1 次调整达成舍位平衡的。
- 2.上面的情况往往是混合出现的。
- 3.可以先处理第(1)种情况，即所有非合计行列平衡差符号相同的情况，再处理第(2)种情况，将非合计行/列中不同符号的平衡差消除。全部调整完毕，非合计行与非合计列的平衡差只能各为一种符号。此时再处理第(3)种和第(4)种情况，将非合计行/列的平衡差与合计行/列的平衡差配合消除。最后，如果两个方向行/列的平衡差仍未消除，再按照第(5)中情况处理。这样，就可以对一般性的数据组合完成双向舍位平衡处理了。

第9章 总结

9.1 请总结本次实验的收获

9.2 请给出对本次实验内容的建议

注：本章为酌情加分项。

参考文献

为完成本次实验你翻阅的书籍与网站等

- [1] 林来兴. 空间控制技术[M]. 北京: 中国宇航出版社, 1992: 25-42.
- [2] 辛希孟. 信息技术与信息服务国际研讨会论文集: A 集[C]. 北京: 中国科学出版社, 1999.
- [3] 赵耀东. 新时代的工业工程师[M/OL]. 台北: 天下文化出版社, 1998 [1998-09-26]. <http://www.ie.nthu.edu.tw/info/ie.newie.htm> (Big5) .
- [4] 谌颖. 空间交会控制理论与方法研究[D]. 哈尔滨: 哈尔滨工业大学, 1992: 8-13.
- [5] KANAMORI H. Shaking Without Quaking[J]. Science, 1998, 279 (5359): 2063-2064.
- [6] CHRISTINE M. Plant Physiology: Plant Biology in the Genome Era[J/OL]. Science , 1998 , 281 : 331-332[1998-09-23]. <http://www.sciencemag.org/cgi/collection/anatmorp>.