哈爾濱工業大學

实验报告

实 验（二）

题　　目 DataLab 数据表示

专　　业 计算学部

学　　号 1190200910

班　　级 1903012

学　　生 严幸

指 导 教 师 史先俊

实 验 地 点 G709

实 验 日 期 2021/3/31

计算机科学与技术学院

目 录

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/优麒麟 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

1) 上实验课前，必须认真预习实验指导书（PPT 或 PDF）
2) 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。
3) 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)

答：

Windows 32 位：char:1 Byte, short:2 Byte, int:4 Byte, long:4 Byte, float:4 Byte,

double:8 Byte, long long:8 Byte, long double:8 Byte, 指针:4 Byte

Windows 64 位：char:1 Byte, short:2 Byte, int:4 Byte, long:4 Byte, float:4 Byte, double:8 Byte, long long:8 Byte, long double:8 Byte, 指针:8 Byte

Linux 32 位：char:1 Byte, short:2 Byte, int:4 Byte, long:4 Byte, float:4 Byte, double:8 Byte, long long:8 Byte, long double:12 Byte, 指针:4 Byte

Linux 64 位：char:1 Byte, short:2 Byte, int:4 Byte, long:8 Byte, float:4 Byte, double:8 Byte, long long:8 Byte, long double:16 Byte, 指针:8 Byte

4) 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 （linux-x64）

答：int 在 47 时出错，long 在 93 时出错，unsigned int 在 48 时出错，unsigned long 在 94 时出错

先用递归程序实现，会出现什么问题？

答：递归公式：f(n) = f(n-1) + f(n-2)，n>=3; f(1) =f(2)=1;问题：当计算到 fib(43)时速度明显变慢，递归层次过深，时间复杂度太高，同时递归层次过深可能导致内存不足。

再用循环方式实现。

答：循环代码：a[1] = a[2] = 1; for(int i = 3; i < n; i++) a[i] = a[i-1]+a[i-2];

5) 写出 float/double 类型最小的正数、最大的正数（非无穷）

答：float 类型最小的正数：2^{-149}，1.401298464324817E-45

float 类型最大的正数：$(2-2^{-23})\times 2^{127}$，3.4028234663852886E+38

double 类型最小的正数：2^{-1074}，5E-324

double 类型最大的正数：$(2-2^{-52})\times 2^{1023}$，1.7976931348623157E+308

6) 按步骤写出 float 数-10.1 在内存从低到高地址的字节值-16 进制

答：-10.1<0,故符号位是 1

将十进制 10.1 转化成二进制：

1010.0001100110011001100110011001100110011001100110011……

转换成科学计数法：

1.0100001100110011001100110011001100110011001100110011E+3

阶码部分=(3+*bias*)=3+127=130=0b 1000 0010

尾数部分保留 23 位：

01000011001100110011010

组合之后(符号位是 1)：

1100 0001 0010 0001 1001 1001 1001 1010

即 C1 21 99 9A，由于在内存中采用小端存储方式，实际存储内容为：

9A 99 21 C1

7) 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（表示的浮点数个数/区域长度）

最大密度：阶码为 0 或 1，实际指数为-126，表示范围为**$-(2-2^{-23})\times 2^{-126}$~$(2-2^{-23})\times 2^{-126}$**，共 2^{25} 个(包括正数、负数、正零、负零)浮点数，密度为 2^{-149}

最小密度：阶码为 254，实际指数为 127，表示范围为 1×2^{127}~$(2-2^{-23})\times 2^{127}$，共 2^{23} 个浮点数，密度为 $2^{127}\times 2^{-23}=2^{104}$。或者是这个区域关于原点对称的在负半轴上的那个区域，密度相同。

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

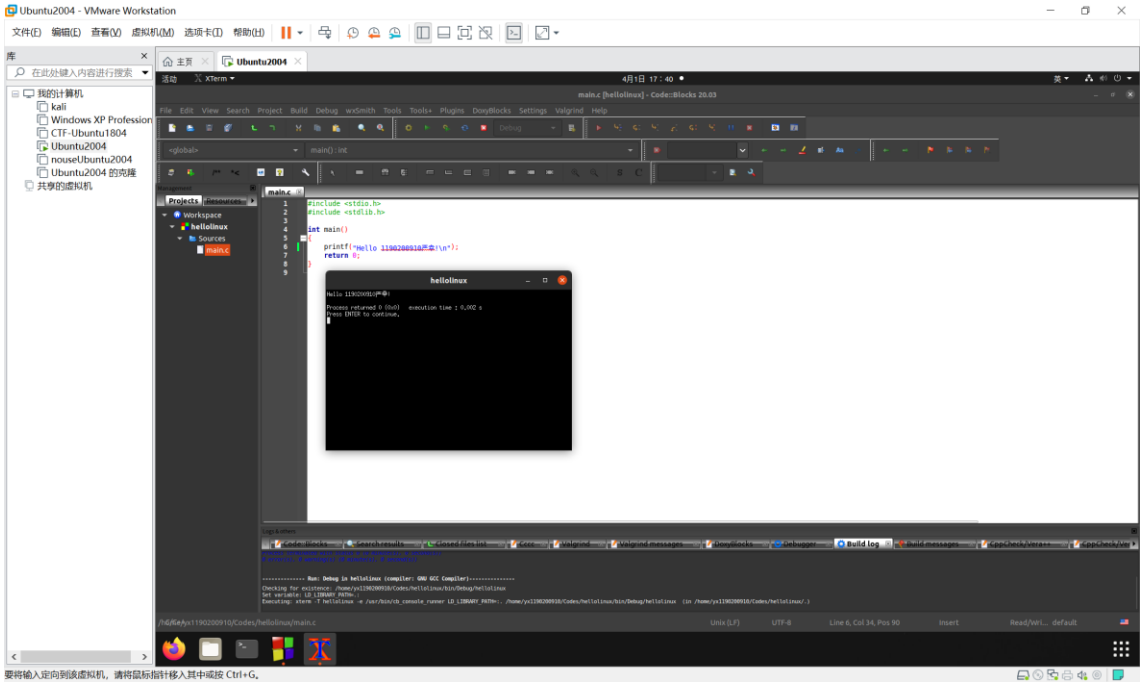


图 2-1　Ubuntu 下 CodeBlocks 截图

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。

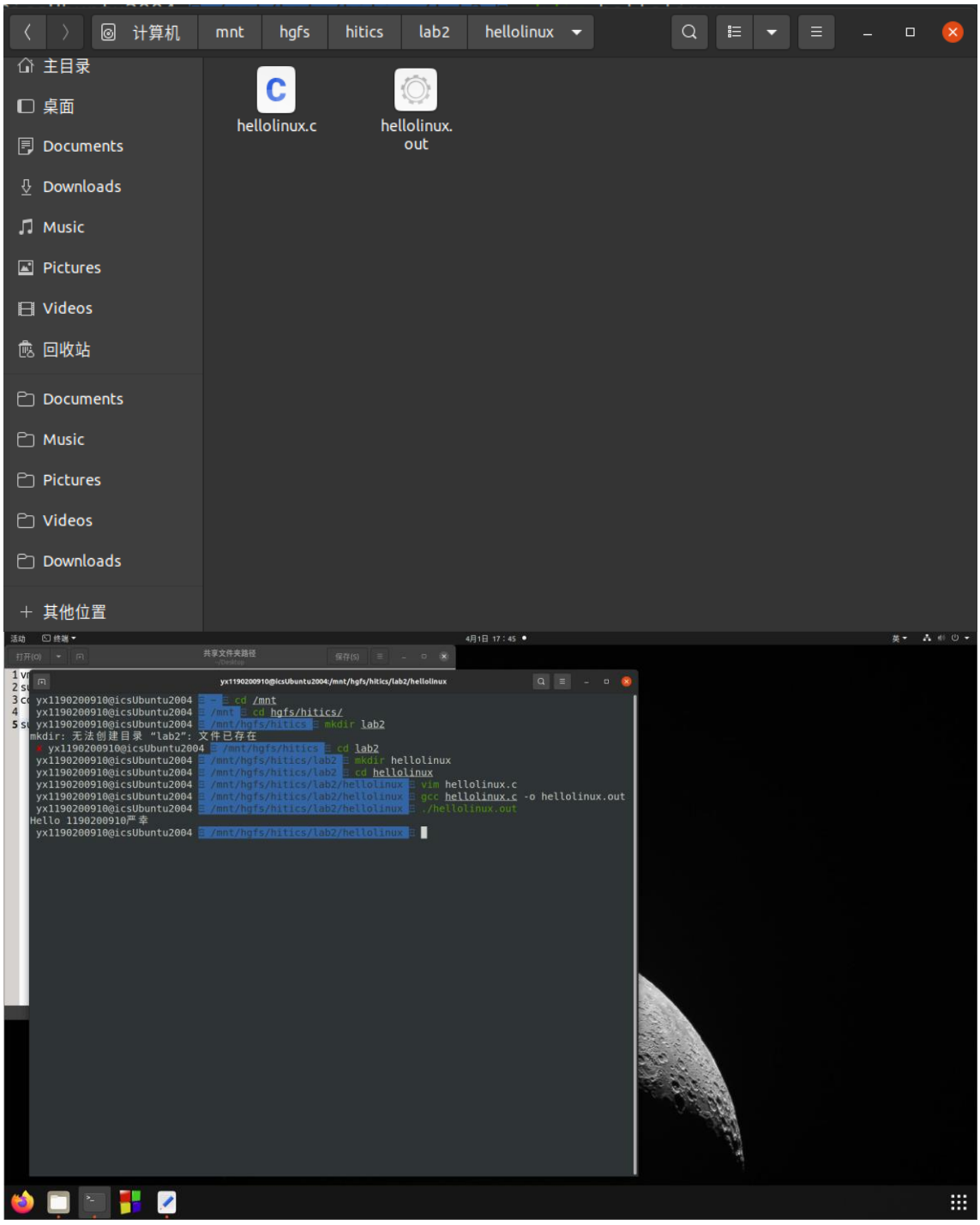


图 2-2　Ubuntu 与 Windows 共享目录截图

第 3 章 C 语言的数据类型与存储

3.1 类型本质

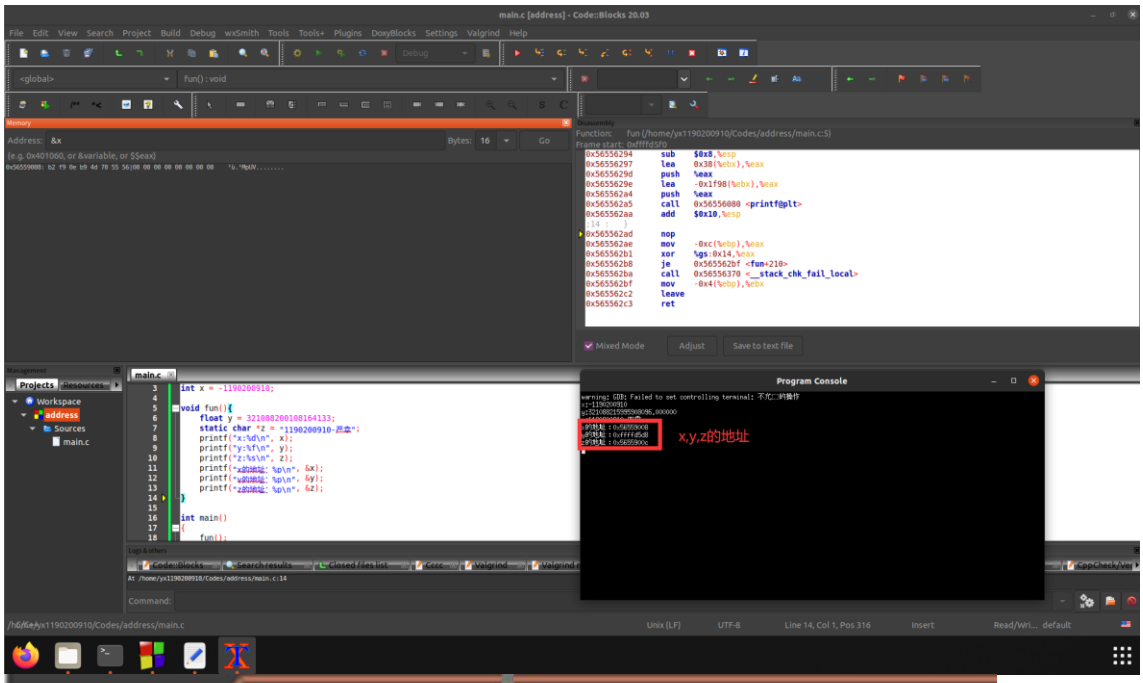
	Win/VS/x86	Win/VS/x64	Win/CB/32	Win/CB/**64**	Linux/CB/32	Linux/CB/64
char	1	1	1	1	1	1
short	2	2	2	2	2	2
int	4	4	4	4	4	4
long	4	4	4	8	4	8
long long	8	8	8	8	8	8
float	4	4	4	4	4	4
double	8	8	8	8	8	8
long double	8	8	12	16	12	16
指针	4	8	4	8	4	8

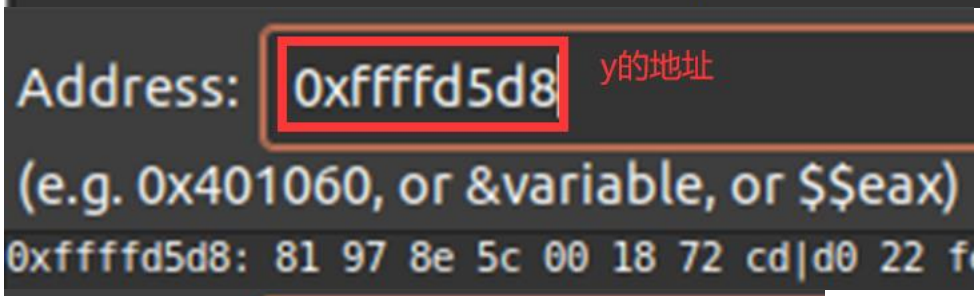
C 编译器对 sizeof 的实现方式：____在编译期就确定值的大小，汇编后变成一个立即数，执行时相当于常数。____

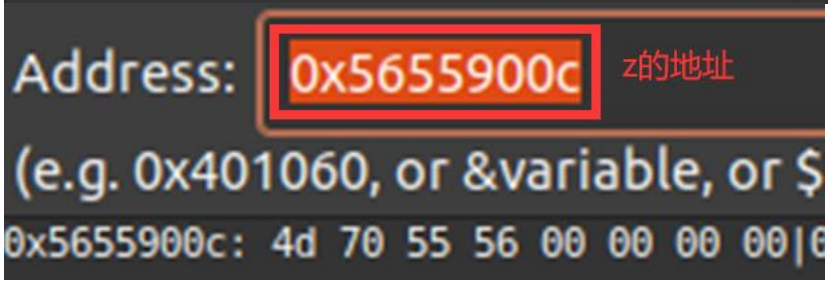
3.2 数据的位置-地址

打印 x、y、z 输出的值：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

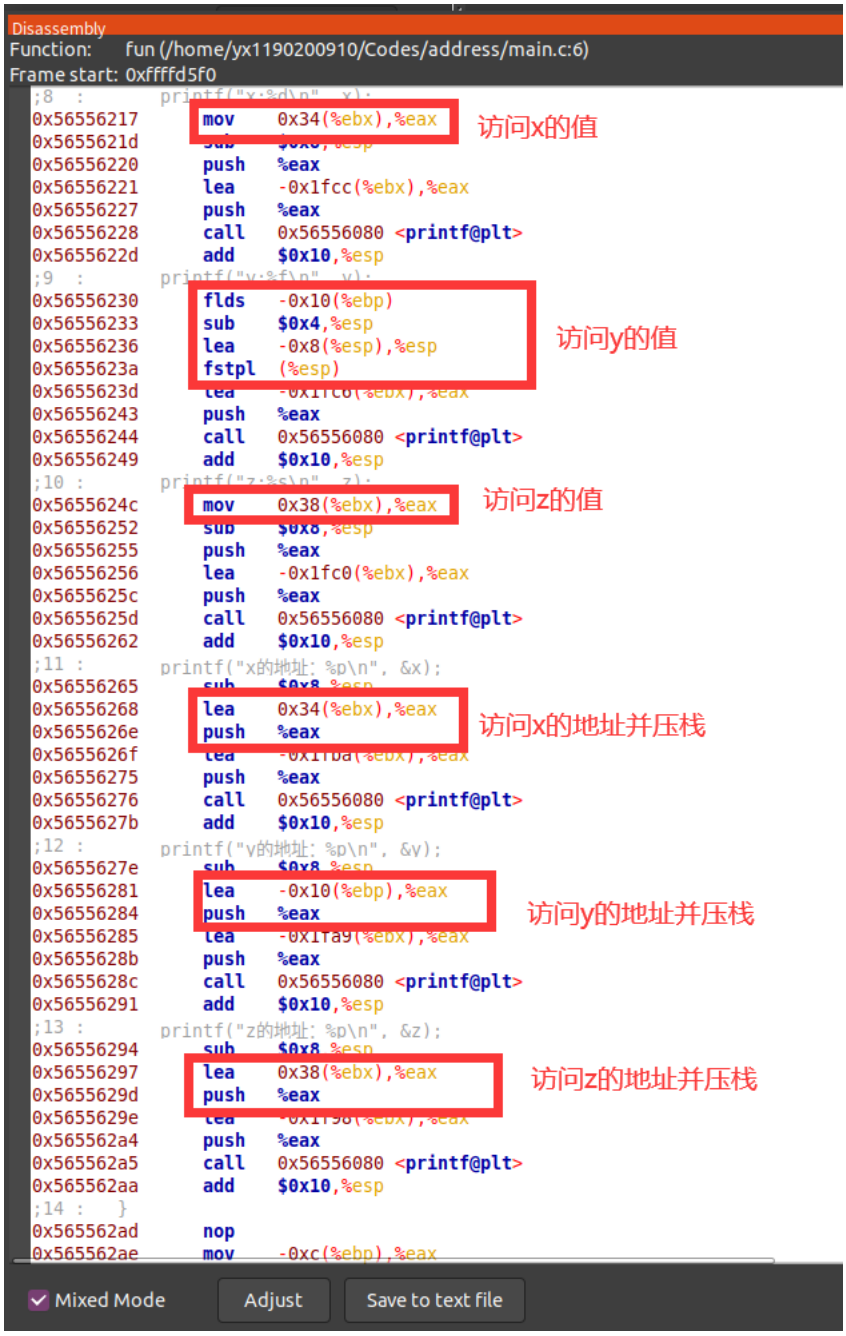






(0x56559008) x 每字节的内容: b2 f9 0e b9
(0xffffd5d8)　y 每字节的内容: 81 97 8e 5c
(0x5655900c) z 每字节的内容: 4d 70 55 56

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, y, z 均通过内存地址访问，局部变量通过堆栈传参。

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：___数值型常量存储在常量区，或者以立即数的形式存储在代码段。变量存储要么保存在内存的动态数据区，要么保存在寄存器当中，这些变量在程序或函数结束会被销毁。___

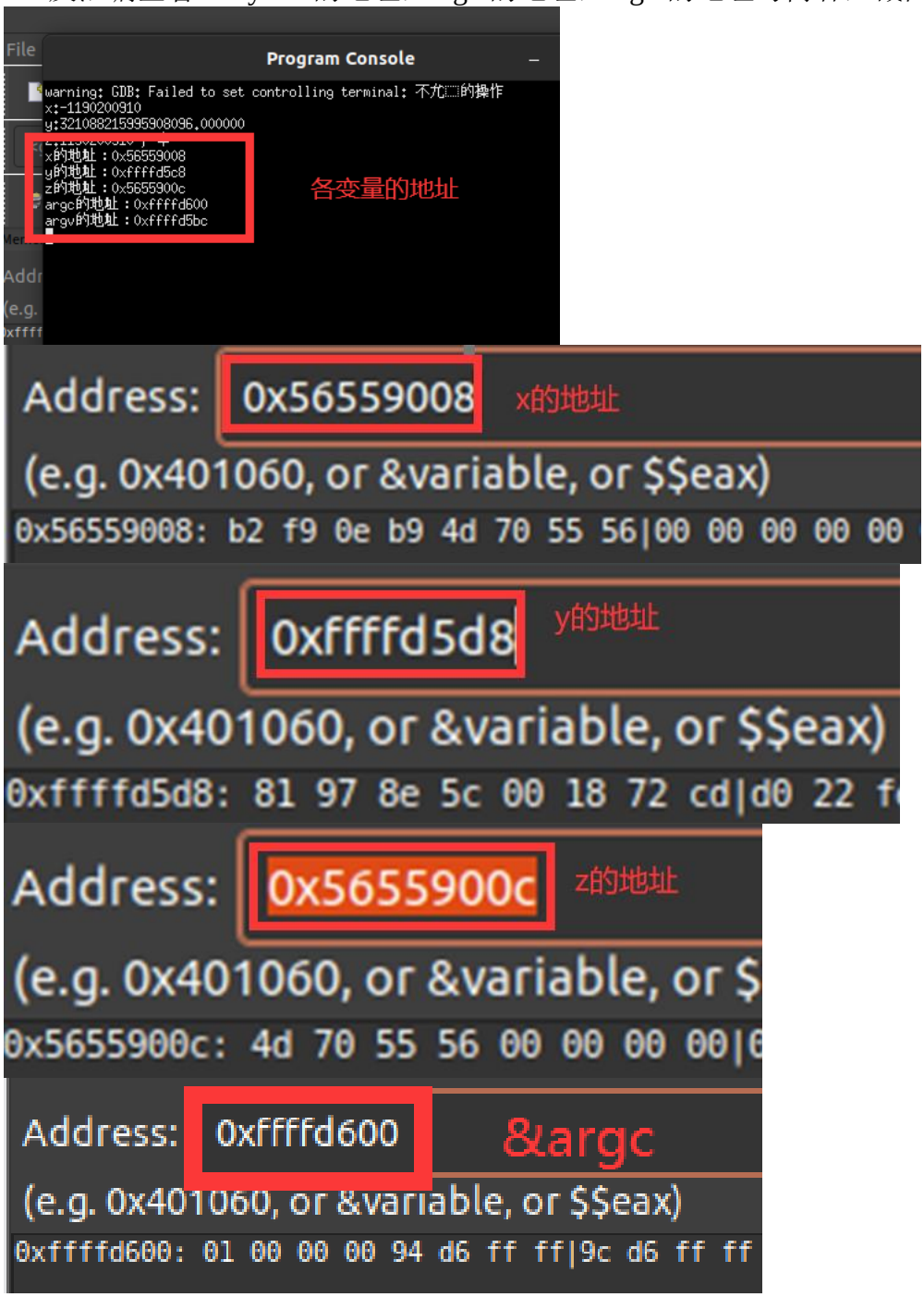
字符串常量与变量在存储空间上的区别是：__字符串常量储存在常量区，只可读不可写。字符串变量储存在静态全局初始化区，和全局变量一起储存在数据区。__

常量表达式在计算机中处理方法是：__在编译后就以立即数或者立即数表达式

的形式存储在代码段，一旦编译即不可修改。__

3.3 main 的参数分析

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



(0x56559008) x 每字节的内容: b2 f9 0e b9
(0xffffd5d8)　y 每字节的内容: 81 97 8e 5c
(0x5655900c) z 每字节的内容: 4d 70 55 56
(0xffffd600) argc 每字节的内容: 01 00 00 00
(0xffffd5bc) argv 每字节的内容: 94 d6 ff ff
分析：argc 和 argv 都是 main 函数的参数，都是通过堆栈传参，由调用 main 函数的那个程序压栈，放在堆栈中“函数参数”的那个区域。将 x y z 传递给 main 函数并打印的效果如下图所示。

```
yx1190200910@icsUbuntu2004 ~/Desktop/test ./a.out x y z
argc:4
argc[0]:./a.out
argc[1]:x
argc[2]:y
argc[3]:z
yx1190200910@icsUbuntu2004 ~/Desktop/test
```

3.4 指针与字符串的区别

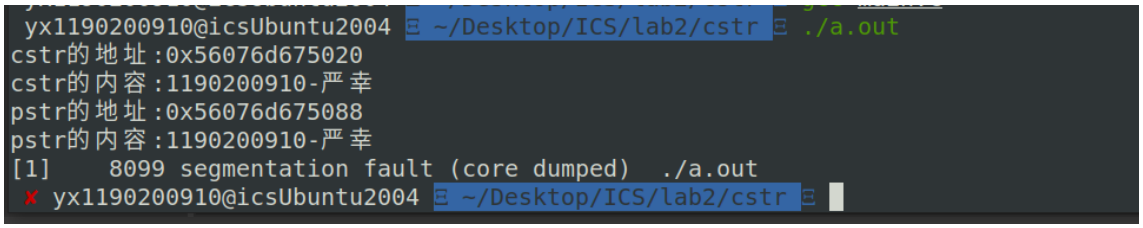
cstr 的地址与内容截图，pstr 的内容与截图，截图 5

```c
#include <stdio.h>
#include <string.h>

char cstr[100]="1190200910-严幸";
char *pstr="1190200910-严幸";

int main(){
    printf("cstr的地址:%p\n", &cstr);
    printf("cstr的内容:%s\n", cstr);
    printf("pstr的地址:%p\n", &pstr);
    printf("pstr的内容:%s\n", pstr);
    strcpy(cstr, "3210882001081641 33");
    strcpy(pstr, "3210882001081641 33");
    printf("cstr:%s\n", cstr);
    printf("pstr:%s\n", pstr);
    return 0;
}
```

本程序的代码。



可以看到当试图向 pstr 字符串复制时程序崩溃。

pstr 修改内容会出现什么问题__strcpy 时会出现 segmentation fault，无法复制，因为 pstr 是字符串常量，处在常量区，只可读不可写。__

第 4 章 深入分析 UTF-8 编码

4.1 提交 utf8len.c 子程序

见附件。

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

strcmp 按照每个汉字的 UTF-8 编码的大小进行排序。既不是按汉字的拼音顺序，也不是按笔画顺序，而是按汉字在 UTF-8 码表的顺序进行排序，先比较前面的汉字，再比较后面的汉字。

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？

需要制作一张根据每个汉字笔画排序已经排好的码表，每个汉字根据其笔画排序有一个优先级，笔画排序靠前的优先级较高，该表将每个汉字的 UTF-8 编码映射为一个表示优先级的正整数。实际应用中对汉字进行排序时，需先根据汉字的 UTF-8 编码到这个表中映射找到它的优先级，再根据优先级对汉字排序。不能直接根据 UTF-8 编码排序。

第 5 章 数据变换与输入输出

5.1 提交 cs_atoi.c

见附件。

5.2 提交 cs_atof.c

见附件。

5.3 提交 cs_itoa.c

见附件。

5.4 提交 cs_ftoa.c

见附件。

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：

read 函数和 write 函数的原型如下：

int read(int fd, const void *buf, size_t length)

int write(int fd, const void *buf, size_t length)

其中，fd 都是文件描述符，buf 是缓冲域指针，length 是读取的字节数。

因此，read 函数和 write 函数是按字节读取二进制数据的，读取到一定的字节数即停止，缓冲区使用的指针类型是 void*型，因此这两个函数与数据类型无关，只需要根据不同的数据类型指定不同的读取字节数 length 即可。

第 6 章 整数表示与运算

6.1 提交 fib_dg.c

见附件。出现的问题：当计算到 fib(43)时速度明显变慢，递归层次过深，时间复杂度太高，同时递归层次过深可能导致内存不足。

6.2 提交 fib_loop.c

见附件。

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

```c
#include <stdio.h>

int main(){
    int x = 10;
    int y = x / 0;
    printf("%d\n", y);
    return 0;
}
```

整数除以 0 的 C 语言代码

```
yx1190200910@icsUbuntu2004 ⊟ ~/Desktop/ICS/lab2/divided0 ⊟ vim test1.c
yx1190200910@icsUbuntu2004 ⊟ ~/Desktop/ICS/lab2/divided0 ⊟ gcc test1.c -o test.out
test1.c: In function ‘main’:
test1.c:5:12: warning: division by zero [-Wdiv-by-zero]
    5 |  int y = x / 0;
      |          ^
yx1190200910@icsUbuntu2004 ⊟ ~/Desktop/ICS/lab2/divided0 ⊟ ./test.out
[1]    3518 floating point exception (core dumped)  ./test.out
✘ yx1190200910@icsUbuntu2004 ⊟ ~/Desktop/ICS/lab2/divided0 ⊟
```

这段代码在编译时编译器会给出一个 Warning，因为代码中出现了显式的除以 0 操作，但是仍能编译通过。编译后的程序在运行时出错，报出了 **floating point exception (core dumped)**，程序并没有正常执行，直接崩溃停止。

除以极小浮点数，截图：

```c
#include <stdio.h>

int main(){
    double x = 1234567;
    double y = x / 0.00000000000000000000000000000000000000000000000001;
    printf("%f\n", y);
    return 0;
}
```

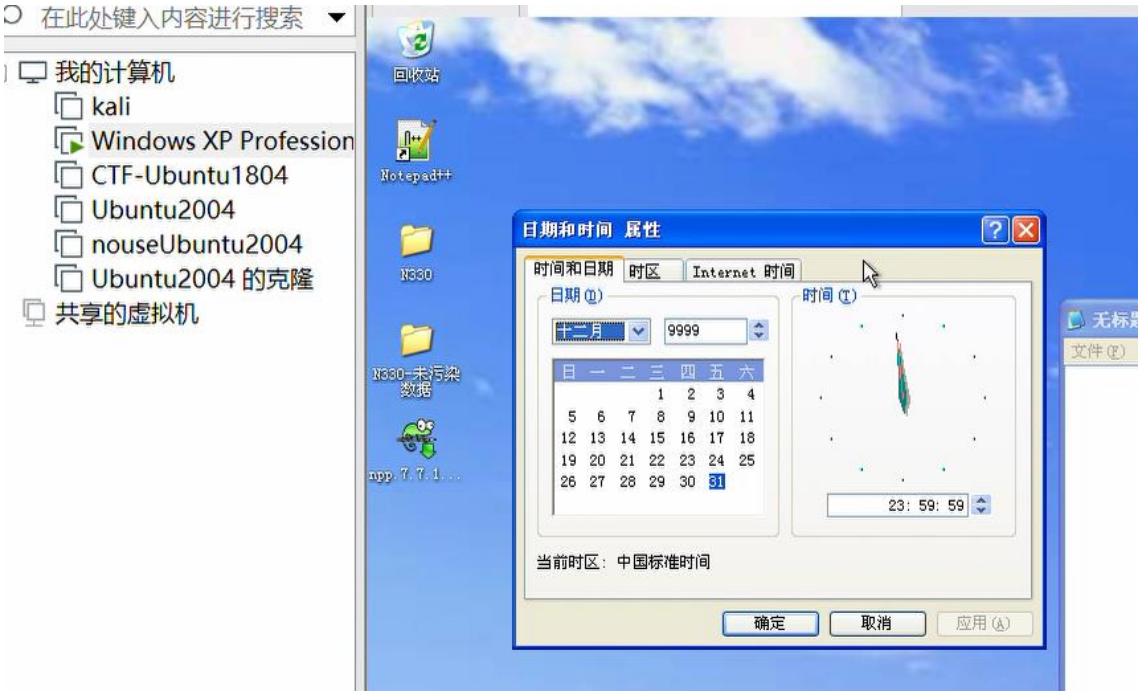
除以极小浮点数的代码

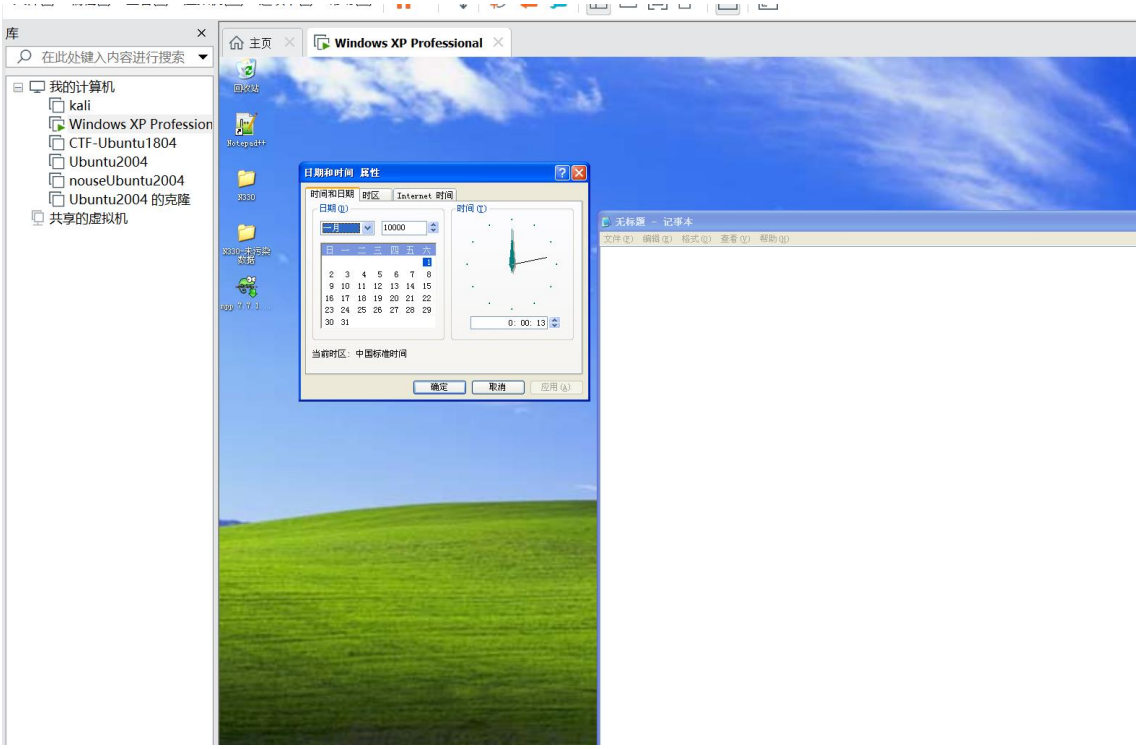
```
yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab2/divided0 vim test2.c
yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab2/divided0 gcc test2.c -o test.out
yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab2/divided0 ./test.out
12345669999999999814117021104661787351319333266102681 6.000000
yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab2/divided0
```

这段编译后既没有 warning 也没有 error，运行后 1234567 除以一个很接近 0 的数精度丢失了很多。

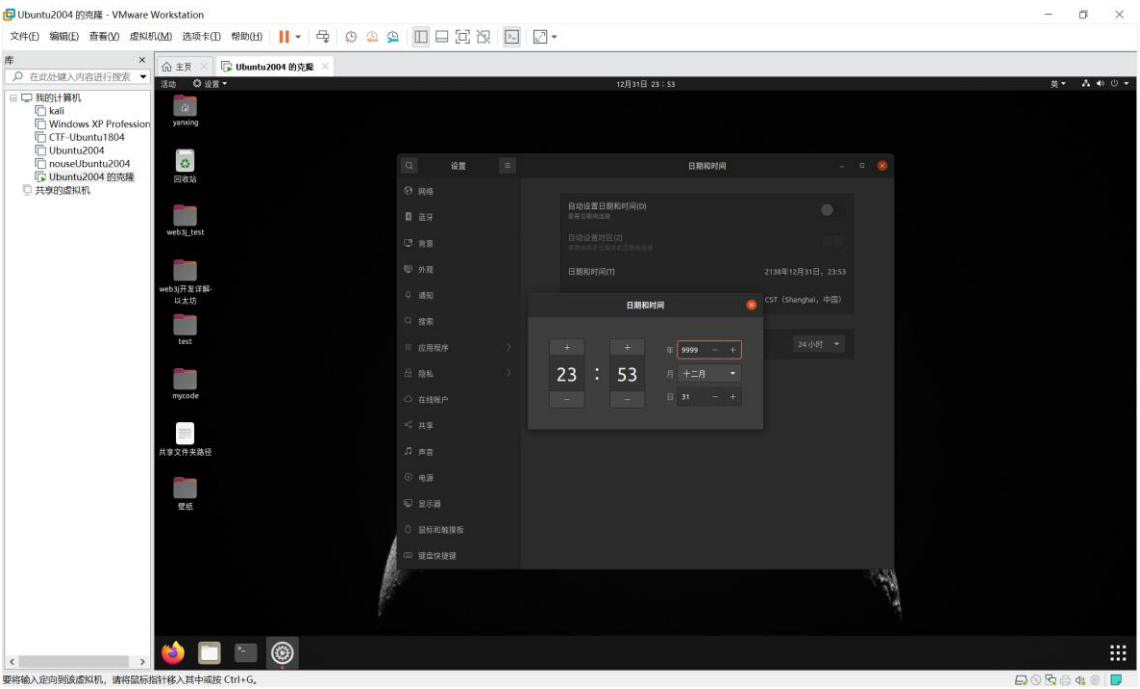
6.5 万年虫验证

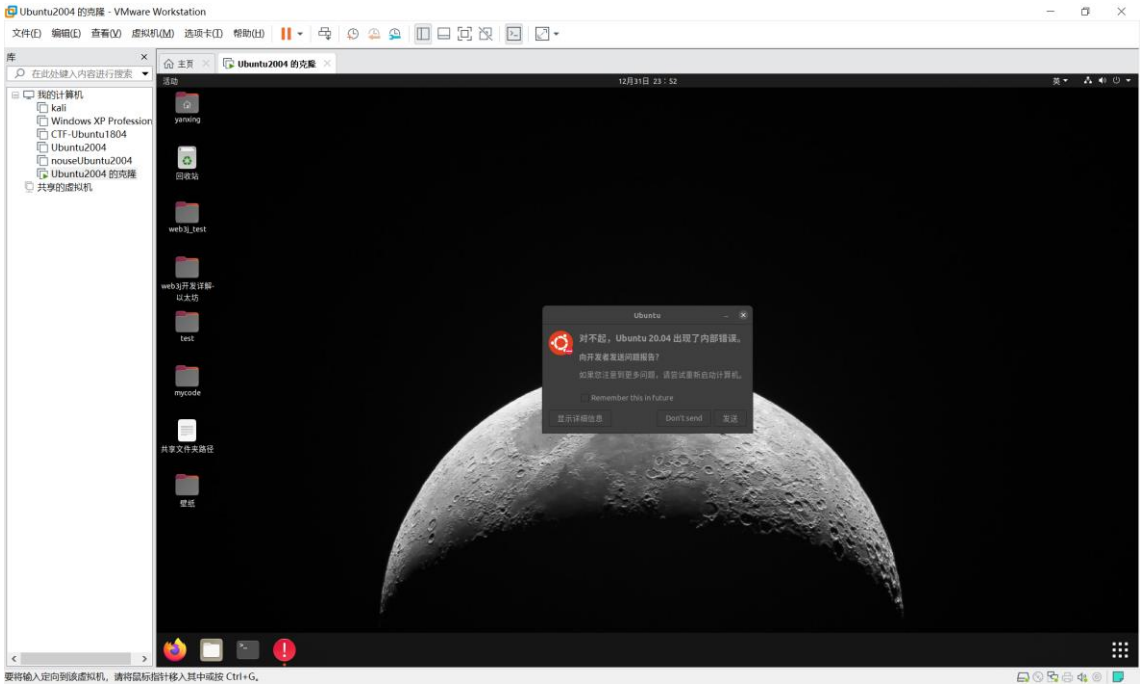
你的机器到 9999 年 12 月 31 日 23:59:59 后，时钟怎么显示的？Windows/Linux 下分别截图：





当时间进行到 9999 年 12 月 31 日 23:59:59 时，电脑突然卡顿一下。之后显示到了 10000 年 1 月 1 日 0:00:13 秒，电脑异常卡顿，已经无法正常使用。将时间改回到 2099 年之后电脑又恢复正常响应。



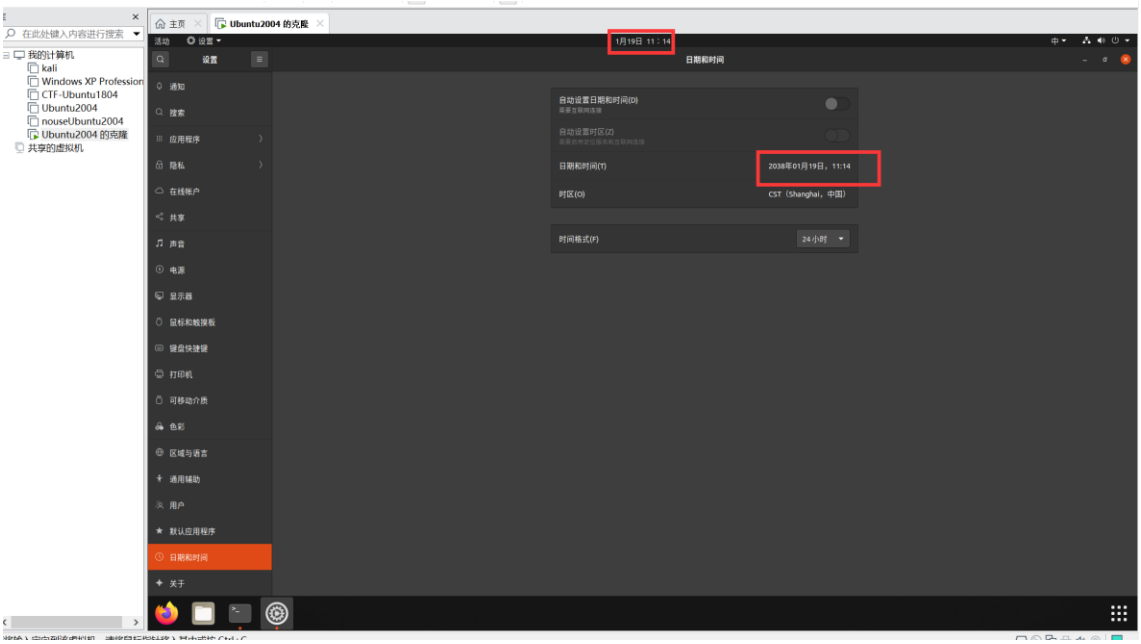


Linux 下试图修改时间到 9999 年，然后系统报错强制终止这个行为。

6.6 2038 虫验证

2038 年 1 月 19 日中午 11:14:07 后你的计算机时间是多少，Windows/Linux 下分别截图

Windows XP 顺利度过了 2038 虫，时间仍然正常显示，电脑其他程序未出现崩溃反应，



Ubuntu 也顺利通过了 2038 虫考验，时间正常显示。

第 7 章 浮点数据的表示与运算

7.1 手动 float 编码：

按步骤写出 float 数-10.1 在内存从低到高地址的字节值（16 进制）。
答：-10.1<0,故符号位是 1
将十进制 10.1 转化成二进制：
1010.0001100110011001100110011001100110011001100110011……

转换成科学计数法：
1.01000011001100110011001100110011001100110011001100110011E+3

阶码部分=(3+*bias*)=3+127=130=0b 1000 0010

尾数部分保留 23 位：
01000011001100110011010

组合之后(符号位是 1)：
1100 0001 0010 0001 1001 1001 1001 1010

即 C1 21 99 9A
由于在内存中采用小端存储方式，实际存储内容为：(由低到高)
9A 99 21 C1

编写程序在内存验证手动编码的正确性，截图。

```c
#include <stdio.h>

int main(){
    float f = -10.1;
    unsigned char *p = &f;
    for(int i = 0; i < 4; i++){
        printf("%x ", *p);
        p++;
    }
    return 0;
}
```

```
 yx1190200910@icsUbuntu2004 ~/Desktop/test  vim test.c
 yx1190200910@icsUbuntu2004 ~/Desktop/test  gcc test.c
test.c: In function ‘main’:
test.c:5:21: warning: initialization of ‘unsigned char *’ from incompatible pointer type ‘flo
-Wincompatible-pointer-types]
    5 |  unsigned char *p = &f;
      |                     ^
 yx1190200910@icsUbuntu2004 ~/Desktop/test  ./a.out
9a 99 21 c1 %
 yx1190200910@icsUbuntu2004 ~/Desktop/test
```

由此程序的输出可知上述手动转换正确。

7.2 特殊 float 数据的处理

提交子程序 floatx.c，要求：

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.c 程序见附件。输出结果如下。

```
 yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab2/float  ./floatx.out
正0:0.000000
负0:-0.000000
最小浮点正数:0.00000000000000000000000000000000000000000000140129846432481707092372958328991613128026
19418765157718
最大浮点正数:340282346638528859811704183484516925440.000000
最小正的规格化浮点数:0.00000000000000000000000000000000000000011754943508222875079687365372222456778186
6655567720875215087 5171
正无穷大:inf
Not a Number: nan
 yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab2/float
```

7.3 验证浮点运算的溢出

提交子程序 float0.c

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.c 程序见附件，输出结果如下。

```
yx1190200910@icsUbuntu2004:~/Desktop/ICS/lab2/float
 yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab2/float  vim float0.c
 yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab2/float  gcc float0.c -o float0.out
float0.c: In function ‘main’:
float0.c:8:15: warning: division by zero [-Wdiv-by-zero]
    8 |  float f0 = f / 0;
      |             ^
 yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab2/float  ./float0.out
浮点数100除以0的结果： inf
浮点数100除以极小浮点数的结果： 99999998848415475373493 4528.000000
```

7.4 类型转换的坑

实验指导 PPT 第 5 步骤的 x 变量，执行 x=(int)(float)x 后结果为多少？

原 x=_-1190200910_，现 x=_ -1190200960 _

7.5 讨论 1：有多少个 int 可以用 float 精确表示

有_$2^{27}+2^{24}$_个 int 数据可以用 float 精确表示。

是哪些数据呢？_是以下数据，这里用 int 的 32 个二进制位表示。其中,x 表示该位可以取 0 或 1，0 表示该位只能取 0，1 表示该位只能取 1。_

float 尾数一共 23 位，因此 int 中以 1 开头的，24 位以内的数都可以用 float 精确表示，其余的 int 能用 float 表示的数都可以通过这些数偏移而来。

零：

0000 0000 0000 0000 0000 0000 0000 0000

这些正数：

0000 0000 0000 0000 0000 0000 0000 0001
0000 0000 0000 0000 0000 0000 0000 001x
0000 0000 0000 0000 0000 0000 0000 01xx
0000 0000 0000 0000 0000 0000 0000 1xxx
0000 0000 0000 0000 0000 0000 0001 xxxx
……
0000 0000 01xx xxxx xxxx xxxx xxxx xxxx
0000 0000 1xxx xxxx xxxx xxxx xxxx xxxx
0000 0001 xxxx xxxx xxxx xxxx xxxx xxx0
0000 001x xxxx xxxx xxxx xxxx xxxx xx00
0000 01xx xxxx xxxx xxxx xxxx xxxx x000
……
01xx xxxx xxxx xxxx xxxx xxxx x000 0000

正数一共是 $2^0+2^1+2^2+\cdots+2^{23}+2^{23}\times 7=2^{26}+2^{23}-1$ 个。

负数是上面正数的相反数，加上一个单独的 -2^{-31}，因此负数一共是 $2^{26}+2^{23}$ 个。

加起来在 int 中一共有 $2^{27}+2^{24}$ 个数能用 float 精确表示。

7.6 讨论 2：怎么验证 float 采用的向偶数舍入呢

基于上个讨论，开发程序或举几个特例用 C 验证即可！

截图与标注说明！

使用的特例(二进制/十进制)：

(舍入)

1.1000 0000 0000 0000 0000 001 0010 → 1.1000 0000 0000 0000 0000 001
1.500000134110450744628906 25 → 1.500000119209289550781 25

(进位)

1.1000 0000 0000 0000 0000 001 1110 → 1.1000 0000 0000 0000 0000 010
1. 500000223517417907714843 75 → 1.5000002384185791015625

(向偶数位舍入)

1.1000 0000 0000 0000 0000 001 1000 → 1.1000 0000 0000 0000 0000 010
1. 500000178813934326171875 → 1. 5000002384185791015625

```c
#include <stdio.h>
int main()
{
    float f1 = 1.500000134110450744628906 25;
    printf("f1=%.23f\n", f1);
    float f2 = 1.500000223517417907714843 75;
    printf("f2=%.23f\n", f2);
    float f3 = 1.500000178813934326171875 ;
    printf("f3=%.23f\n", f3);

    return 0;
}
```

测试代码截图

```
yx1190200910@icsUbuntu2004 ~/Desktop/test vim test.c
yx1190200910@icsUbuntu2004 ~/Desktop/test gcc test.c
yx1190200910@icsUbuntu2004 ~/Desktop/test ./a.out
f1=1.50000011920928955078125
f2=1.50000023841857910156250
f3=1.50000023841857910156250
```

测试代码运行结果，与预期相符，证明 float 确实是向偶数位舍入。

7.7 讨论 3：float 能精确表示几个 1 元内的钱呢

人民币 0.01-0.99 元之间的十进制数，有多少个可用 float 精确表示？是哪些呢？

答：由于人民币的最小单位是“分”，即“0.01 元”，因此仅需考虑两位小数即可。0.01 元-0.99 元中有 3 个数可以用 float 精确表示，分别是 0.25 元 0.50 元 0.75 元。

7.8 Float 的微观与宏观世界

按照阶码区域写出 float 的最大密度区域的范围及其密度，最小密度区域及其密度（区域长度/表示的浮点个数）：_最大密度区域$-(2-2^{-23})\times 2^{-126}$~$(2-2^{-23})\times 2^{-126}$_、_最大密度 2^{-149}(能表示的最接近的两个浮点数之间的间隔)_、_1×2^{127}~$(2-2^{-23})\times 2^{127}$_、_$2^{104}$_

微观世界：能够区别最小的变化_2^{-149}_，其 10 进制科学记数法为__1.401E-45__

宏观世界：不能区别最大的变化__2^{104}-1__，其 10 进制科学记数法为_2.028240E+31_

7.9 讨论：浮点数的比较方法

从键盘输入或运算后得到的任意两个浮点数，论述其比较方法以及理由。

比较两浮点数：用两数做差，比较这个差的绝对值和一个非常小的常数 EPS(一般定义为 1E-7)，如果小于 EPS，即认为两数相等。否则再看差值的正负，如果是

正的则认为前数大于后数，否则小于。

由于浮点数到了计算机中表示精度可能有丢失，因此不能用“比较浮点数内存中的每一位是否相等”来判断两数是否相等。

第 8 章 舍尾平衡的讨论

8.1 描述可能出现的问题

传统的舍尾处理往往采用简单的四舍五入算法，这样在计算数字之和时如果很多数字都同时采用“五入”或同时采用“四舍”保留到一定的小数位，做出的报表精度会有很大损失。例如，如果原始的数据是 4.5+4.5=9.0，在采用四舍五入保留到整数位之后，做出的报表看起来就像是 5+5=9，这显然是荒谬的，导致这样问题的原因就是 4.5 和 4.5 同时采用了“五入”，而结果 9.0 采取了“四舍”，这样的每个数字丢失精度带来的一点点小误差在慢慢累积之后可能在宏观上显示出较大额的误差。

再一例：50.23+5.24=55.47，直接给每个数保留一位小数之后变成 50.2+5.2=55.5，出现偏差。

事实上，在税务计算中，经常涉及到单位换算(万元→千元、千元→元)，因此这样的偏差十分常见。

8.2 给出完美的解决方案

注：本部分给出的方案参考了 CSDN 博客[7]。

方法 1：将计算的偏差直接加到第一个数据上，只用第一个数据进行修正。例如，对算式“2.2+3.3+4.4+5.2+6.3+7.4+8.2+9.3+10.4=56.7”进行保留到整数的操作时，如果直接采用四舍五入会得到错误的算式“2+3+4+5+6+7+8+9+10=57”。最简单的修正方法是：等式左边的正确运算结果应该是 54，与等式右边的数 57 产生了大小为 3 的偏差，修正时将这个偏差 3 直接加到等式左边第一个数“2”上，修正后的算式变成“5+3+4+5+6+7+8+9+10=57”，看起来似乎变得合理了一些，因为等式确实平衡了。但是这样带来的问题是，只用一个数来承担偏差的修正，这个数修正后与修正前的变化过大，财务报表做出来之后，这一项的数值可能无法反映企业的真实情况。比较好的做法是，将偏差加在等式左边绝对值最大的一项数值上，这样一来，修正前后这一项数值的“相对变化量”就小了很多。例如在上面的例子中，如果用等式左边的 2 来修正，那么修正前后这一项数值的相对变化量为“3/2=150%”，而如果用绝对值最大的 10 来修正，那么相对变化量为“3/10=30%”，这样的修正方法又合理一些。但是还有更好的修正方式，就是将偏差分摊在多个数值上，我们将在下一段“方法 2”当中讨论。

方法 2：在阐述方法 2 前，先介绍“最小调整值”的概念。最小调整值，就是舍位后最小精度的单位值。例如保留到整数，最小调整值就是±1；保留到一位小数，最小调整值就是±0.1，以此类推。方法 2 是利用算式中的多个数据共同分摊偏差，每个数据最多分摊的偏差大小就是最小调整值。例如，保留到整数时，每个数就最多承担±1 的偏差，这样可以保证每一项数值在调整前后的变化都比较

小，不会对做出的报表上某一项数值的理解产生较大偏差。具体来讲，本方法是将算式左边的所有项按绝对值从大到小排序，然后取其中绝对值较大的那些项，每一项都承担大小为“最小调整值”的偏差。例如上面的算式“2+3+4+5+6+7+8+9+10=57”，等式左右产生的总偏差为+3，将这+3 的误差分成三份，每一份是+1 的偏差，再将这三份偏差加到等式左边绝对值最大的 3 项数值上，修正后的算式为“2+3+4+5+6+7+9+10+11=57”，这时，等式左边数值的最大变化量为“1/8=12.5%”，大大减小了报表中数值的变化程度，使得报表中的数值能够最大程度地反映真实情况。下面将使用几个例子来阐述本方法。

例 1：原始算式为“1.48+1.42+0.32+6.48+0.98+1.39=12.07”，直接保留整数得到算式“1+1+0+6+1+1=12”，最小调整值为±1，等式左边的运算结果为 10，12 与它产生了+2 的偏差，将总偏差分为两份大小为+1 的偏差，加到等式左边绝对值最大的 6 和 1 两项数值上，得到修正后的算式“2+1+0+7+1+1=12”。

例 2：原始算式为“1.34+7.53+3.12+8.98=20.97”，直接保留一位小数得到算式“1.3+7.5+3.1+9.0=21.0”，最小调整值为±0.1，等式左边的运算结果为 20.9，21.1 与其产生了+0.1 的偏差，这 0.1 的偏差变为 1 份，加到等式左边绝对值最大的一项 9.0 数值上，得到修正后算式“1.3+7.5+3.1+9.1=21.0”。

第 9 章 总结

9.1 请总结本次实验的收获

- 第一次真正的在自己的电脑上感受到了千年虫的危害，学会了电脑遇到千年虫后的修复方式。
- 学会了使用 GDB 调试程序、objdump 反汇编
- 了解了不同类型变量在内存中保存的区域
- 了解了舍位平衡的概念

9.2 请给出对本次实验内容的建议

- 千年虫部分的实验难度比较大，建议老师可以添加一些提示信息。
- 实验的 PPT 部分有些句子表达比较模糊。

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.

[7] raqsoft.CSDN 原创博客，https://blog.csdn.net/raqsoft/article/details/83503757