

作业 1：数据的表示

17307130178 宁晨然

一、概述

本次作业重心在于对于计算机中整数、浮点数的机内表示和运算过程，通过观察与常识不符合的程序结果来探索机器运算方式。以下分为实验过程、实验代码、实验感受介绍。

二、实验过程

1. 下述两个结构所占存储空间多大？结构中各分量所在位置相对于结构起始位置的偏移量是多少？请编写程序以验证你的答案。若使用 `#pragma pack(2)` 语句，则结果又如何？若使用 `#pragma pack(32)` 语句，结果又如何？

答：

为了查找到每个类型在 `struct` 中的偏移位置，我使用了 `&(((s*)0)->m)` 语句，用来获得每个类型的地址偏移量。

(1) `test1` 结构所占存储空间 40，`test2` 结构所占存储空间 48。

`test1` 中各分量所在位置的偏移量分别为：0,18,24,32

`test2` 中各分量所在位置的偏移量分别为：0,18,24,32

```
test1:
size of test1= 40
positions of x1,x2,x3,x4 are: 0,18,24,32

test2:
size of test2= 48
positions of x1,x2,x3,x4 are: 0,18,24,32
```

```
struct test1
{
    char x1[17];
    short x2[2];
    int x3;
    long long x4;
};
```

原因是：

`test1` 和 `test2` 的结构中虽然是分别占用 17,4,4,8,若按照节省空间的方法存放，只需要空间 31 即可，但是系统默认空间存储方式中有对齐操作，是为了方便用户和机器查找所需要的数据的地址，更便于查找。默认对齐操作中，对齐方式（变量存放的起始地址相对于结构的起始地址的偏移量），有规则：Char 偏移量必须为 `sizeof(char)` 即 1 的倍数；int 偏移量必须为 `sizeof(int)` 即 4 的倍数；float 偏移量必须为 `sizeof(float)` 即 4 的倍数；double 偏移量必须为 `sizeof(double)` 即 8 的倍数 Short 偏移量必须为 `sizeof(short)` 即 2 的倍数。

所以在 `test1` 的情况中，`x1` 起始位置 0 占用 17 个字节，然后从 18(2*9)开始存 short 类型 `x2` 占用 4 个字节，放 2 个空字节，再从 24 (4*6) 开始存 int 类型 `x3` 占用 4 个字节（目前有 28 个字节），放 4 个空字节，最后从 32 开始存放 long long 类型 `x4` 占用 8 个字节，共有 40 个字节。这是默认对齐方式，使得 `test1` 的 `struct` 类型占用 40 个字节。

语句`__attribute__((aligned (n)))`的意思是，让所作用的结构成员对齐在 n 字节自然边界上。如果结构中有成员的长度大于 n ，则按照最大成员的长度来对齐。[reference1]

对于 test2 中添加了`__attribute__((aligned(16)))`的语句，意思是 test2 对齐方式满足最大 16 字节对齐。由于 test2 中最大的类型是 long long，字节数为 $8 < 16$ ，所以各类型的起始位置方式跟 test1 一样，但是对于整个 test2 的类型，需要以 16 字节的方式对齐，所以选取了 16 的倍数 48 为 test2 类型的字节数。(相当于 test1 后面多了 8 个空字节)

(2)当使用`#pragma pack(n)`后结果

`#pragma pack(n)`的意思是： n 字节对齐就是说变量存放的起始地址的偏移量有两种情况：第一、如果 n 大于等于该变量所占用的字节数，那么偏移量必须满足默认的对齐方式，第二、如果 n 小于该变量的类型所占用的字节数，那么偏移量为 n 的倍数，不用满足默认的对齐方式。结构的总大小也有个约束条件，分下面两种情况：如果 n 大于所有成员变量类型所占用的字节数，那么结构的总大小必须为占用空间最大的变量占用的空间数的倍数；否则必须为 n 的倍数。[reference2]

```
test1:
size of test1= 34
positions of x1,x2,x3,x4 are: 0,18,22,26

test2:
size of test2= 48
positions of x1,x2,x3,x4 are: 0,18,22,26
```

当 $n=2$ 时。

原因是：

所以根据规则，test1 的对齐方式略有变化。x1 (char 1<2) 占用 1 字节，后填充 1 个空字节，short 类型 x2 占用 2 字节后，共用了 3 字节。由于 int (4>2)，所以对齐方式改变为以 2 对齐，此时 $3 \% 2 = 1$ 已经不对齐，继续填充 x3 4 个字节，至 7 字节。long long 同理($8 > 2$)，以 2 对齐，此时 $7 \% 2 = 1$ 已经不对齐，继续填充 x4 8 个字节，至 15 字节。test1 的总类型字节数也变为 16 字节，因为有成员 int 和 long long 变量占用空间数大于 n ，所以为 2 的倍数即可，此时 16 字节满足条件。

对于 test2，和 test1 类似，各成员的起始位置上面分析一样。关键是总字节数，因为 test2 不仅需要满足`#pragma pack(2)`还要满足`__attribute__((aligned(16)))`，所以总字节数需要是 2 和 16 的倍数，则在 16 字节后添加 14 个字节成为 30 字节对齐。

```
test1:
size of test1= 40
positions of x1,x2,x3,x4 are: 0,18,24,32

test2:
size of test2= 48
positions of x1,x2,x3,x4 are: 0,18,24,32
```

当 $n=32$ 时。

原因是：

相比较于 $n=2$ ，这个 32 使得所有的成员都满足 < 32 的条件，所以在内部的对齐方式`#pragma pack(32)`并没有进行对默认对齐方式改变操作。所以内部的对齐方式与(1)中的默认对齐一样，得到的结果也一样。分析同默认对齐方式一样。

但对于总类型字节数，由于 32 大于结构内类型的最大字节数，结构的总大小必须为占用空间最大的变量占用的空间数的倍数，即 8(long long)的倍数。默认对齐方式 test1 和 test2 的字节数分别是 40 和 48，均为 8 的倍数，所以不需要改变。

最终结果与(1)一样，但是原理略有不同。

2. “-2 < 2”和“-2 < 2u”的结果一样吗？为什么？

答：实验结果如下。两者结果不一样。

```
test1: -2 < 2
answer = 1
test2: -2 < 2u
answer = 0
```

原因是：

整型分为有符号整型与无符号整型。有符号整型的机内表示为二进制补码形式，因为有负数和符号位，为了方便计算全部表示为补码。然而整数后面带上 u 或 U 的是无符号整型，在机内表示为二进制编码。

由于 c 语言中默认的整型常量保存为有符号整数，所以 test1 是两个有符号数的比较，符合自然逻辑数学， $-2 < 2$ 是 true，所以为 1。

test2 中是一个有符号数与无符号数的比较，在 c 语言中遇到运算两边的类型不一致时，会发生隐式类型转换。此处 c 语言会隐式地将有符号数强制类型转换为无符号数，并假设这两个数都是非负的，来执行这个无符号数的比较运算。而 -2 的机内表示为二进制补码，为 fffd，而 2u 二进制表示为 0002，无符号数比较操作时 $\text{fffd} > 0002$ ，故 $-2 < 2u$ 结果为 false，答案为 0。

3. 运行下图中的程序代码，并对程序输出结果进行分析。

```
# include <stdio.h>

void main()
{
    unsigned int a = 3;
    unsigned short b = 3;
    char c = -3;
    int d;

    d = (a > c) ? 1:0;
    printf("unsigned int is %d\n", d);
    d = (b > c) ? 1:0;
    printf("unsigned short is %d\n", d);
}
```

答：输出结果为 0 和 1

```
unsigned int is 0
unsigned short is 1
```

原因是：

unsigned int 是 4 个字节，unsigned short 是 2 个字节。char 类型是 1 个字节，当 char c = -3 时，实际做了一个隐式类型转换，从 -3 有符号数（int 类型补码 4 个字节），进行了位截断操作，只保留了最后 1 个字节，保存在 char 中。

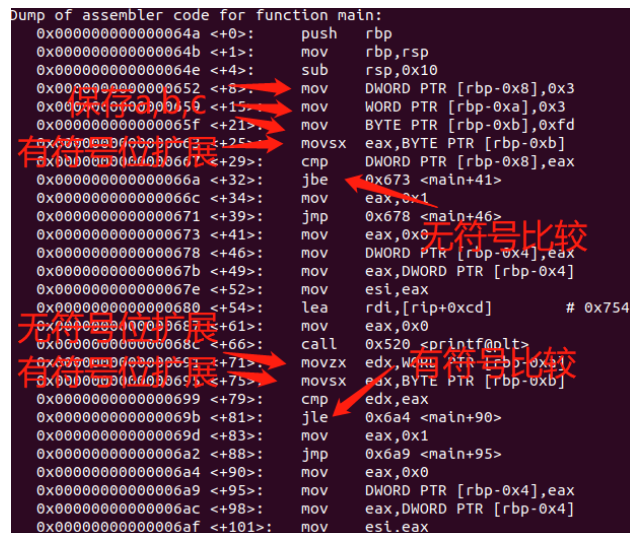
第一个 $a > c$ 是无符号 int 类型与 char 类型的比较，c 语言首先做的操作处理是将 char 类型视为 1 个字节的有符号数进行有符号的扩展，在前方填充 1 使其变成 4 个字节。再用 a 与 c 进行无符号的 int 类型比较，明显 c 大得多，所以 $a > c$ 为 false。

第二个 $b > c$ 是无符号 short 类型与 char 类型的比较，c 语言首先做的处理是将 short 和 char 都位扩展成 4 个字节的 unsigned int 类型，unsigned short 类型需要在前方进行无符号位扩展（加 0），char 类型需要在前方进行有符号位扩展（加 1）。再将两者进行有符号 int 类型比较，所以 $b > c$ 为 true。

具体解释的来源是我查看了反汇编代码，可以从 movzx, movsx 看出是谁进行了位扩展和位截断，又可以从 jle 和 jbe 看出做的是有符号或者无符号数比较。

首先 mov DWORD PTR [rbp-0x8], 0x3 是 unsigned int a=3 操作，后面同理；movsx eax, BYTE PTR [rbp-0xb] 是将 char 类型做有符号的位扩展，并且保存为 4 字节。当 a 和 c

比较时，使用了 jbe，是无符号数之间的比较。后面类似，movzx 是对 short 类型进行无符号位扩展，然后 movsx 对 char 进行有符号位扩展，最后用 jle 用了有符号的 >= 比较。



Dump of assembler code for function main:

```
0x000000000000064a <+0>: push rbp
0x000000000000064b <+1>: mov rbp, rsp
0x000000000000064e <+4>: sub rsp, 0x10
0x0000000000000652 <+8>: mov DWORD PTR [rbp-0x8], 0x3
0x0000000000000659 <+15>: mov WORD PTR [rbp-0xa], 0x3
0x000000000000065f <+21>: mov BYTE PTR [rbp-0xb], 0xfd
0x0000000000000663 <+25>: movsx eax, BYTE PTR [rbp-0xb]
0x0000000000000667 <+29>: cmp DWORD PTR [rbp-0x8], eax
0x000000000000066a <+32>: jbe 0x673 <main+41>
0x000000000000066c <+34>: mov eax, 0x1
0x0000000000000671 <+39>: jmp 0x678 <main+46>
0x0000000000000673 <+41>: mov eax, 0x0
0x0000000000000678 <+46>: mov DWORD PTR [rbp-0x4], eax
0x000000000000067b <+49>: mov eax, DWORD PTR [rbp-0x4]
0x000000000000067e <+52>: mov esi, eax
0x0000000000000680 <+54>: lea rdi, [rip+0xcd] # 0x754
0x0000000000000687 <+61>: mov eax, 0x0
0x000000000000068c <+66>: call 0x520 <printf@plt>
0x0000000000000691 <+71>: movzx edx, WORD PTR [rbp-0xa]
0x0000000000000695 <+75>: movsx eax, BYTE PTR [rbp-0xb]
0x0000000000000699 <+79>: cmp edx, eax
0x000000000000069b <+81>: jle 0x6a4 <main+90>
0x000000000000069d <+83>: mov eax, 0x1
0x00000000000006a2 <+88>: jmp 0x6a9 <main+95>
0x00000000000006a4 <+90>: mov eax, 0x0
0x00000000000006a9 <+95>: mov DWORD PTR [rbp-0x4], eax
0x00000000000006ac <+98>: mov eax, DWORD PTR [rbp-0x4]
0x00000000000006af <+101>: mov esi, eax
```

Annotations in the image:

- 有符号位扩展 (Signed bit extension) points to `movsx` instruction.
- 无符号位扩展 (Unsigned bit extension) points to `movzx` instruction.
- 有符号数比较 (Signed number comparison) points to `jle` instruction.
- 无符号数比较 (Unsigned number comparison) points to `jbe` instruction.

更深层的原因，我推测，C 语言中的二元运算，比如大小比较，需要两边的操作数属于同一个类型，否则会对操作数进行隐式类型转换再行比较。隐式类型转换中会进行位扩展或位截断，比如上述情况中均是扩展到 4 字节再行比较。低类型会转换为高类型进行比较，例如 short 类型与 int 类型进行比较时，会把 short 类型扩展成 int 类型。特殊情况就是两个低类型比较时，比如 short 和 char，会先进行各自的有符号位扩展，再根据情况进行有无符号类型比较。

4. 运行下列代码，并对输出结果进行分析。

```
#include <stdio.h>

void main()
{
    union NUM
    {
        int a;
        char b[4];
    } num;
    num.a = 0x87654321;

    printf("0x%X\n", num.b[1]);
    printf("0x%X\n", num.b[2]);
}
```

答：输出结果为 0x43 0x65

原因是：

首先对 union 和大小端进行解释。

不同于结构成员——它们在结构中都具有单独的内存位置，联合成员则共享同一个内存位置。也就是说，联合中的所有成员都是从相同的内存地址开始。因此，可以定义一个拥有许多成员的联合，但是同一时刻只能有一个成员允许含有一个值。联合让程序员可以方便地通过不同方式使用同一个内存位置。[reference3]

Big-Endian 和 Little-Endian 的定义如下：[reference4]

1) Little-Endian 就是低位字节排放在内存的低地址端，高位字节排放在内存的高地址端。

2) Big-Endian 就是高位字节排放在内存的低地址端，低位字节排放在内存的高地址端。

由于此处是 int 和 char 类型的 union，这个联合一共占用了 4 个字节。并且在 linux 中是小端，所以低地址位存放低位字节，即按顺序存放为 0x21,0x43,0x65,0x87，所以取出 b[1] 和 b[2]时，取得 0x43 和 0x65。

5.请说明下列赋值语句执行后，各个变量对应的机器数和真值各是多少？编写一段程序代码并进行编译，观察默认情况下，编译器是否报 warning。如果有 warning 信息的话，分析为何会出现这种 warning 信息。

```
int a = 2147483648;
int b = -2147483648;
int c = 2147483649;
unsigned short d = 65539;
short e = -32790;
```

答：

```
chty627@ubuntu:~/Documents/homework1$ gcc test1.c -o test1
test1.c: In function 'main':
test1.c:8:2: warning: large integer implicitly truncated to unsigned type [-Woverflow]
  unsigned short d = 65539;
  ^
test1.c:9:2: warning: overflow in implicit constant conversion [-Woverflow]
  short e = -32790;
  ^
```

linux gcc 4.8 中报出 warnings

行	列	单元	信息
5	2	C:\Users\61082\Desktop\temp.cpp	[Warning] this decimal constant is unsigned only in ISO C90
6	2	C:\Users\61082\Desktop\temp.cpp	[Warning] this decimal constant is unsigned only in ISO C90
7	2	C:\Users\61082\Desktop\temp.cpp	[Warning] this decimal constant is unsigned only in ISO C90
		C:\Users\61082\Desktop\temp.cpp	In function 'int main()':
8	21	C:\Users\61082\Desktop\temp.cpp	[Warning] large integer implicitly truncated to unsigned type [-Woverflow]
9	13	C:\Users\61082\Desktop\temp.cpp	[Warning] overflow in implicit constant conversion [-Woverflow]

windows dev c++中报出 warnings

并且两者用 printf("a = %d\nb = %d\nc = %d\nd = %d\ne = %d\n",a,b,c,d,e);输出所有

```
chty627@ubuntu:~/Documents/homework1$ ./test1
a = -2147483648
b = -2147483648
c = -2147483647
d = 3
e = 32746
```

值的结果均为：

原因是：

首先，在 c 语言中的各类型均有精度取值范围。列出如下。

char	-128 ~ +127	(1 Byte)	
short	-32767 ~ + 32768	(2 Bytes)	3×10^4
unsigned short	0 ~ 65536	(2 Bytes)	6×10^4
int	-2147483648 ~ +2147483647	(4 Bytes)	2×10^9
unsigned int	0 ~ 4294967295	(4 Bytes)	4×10^9

前三个 int 类型 a,b,c 均在 int 类型取值边缘，a = 2147483648=0x80000000（机器数）> 2147483647 = 0x7fffffff，而 0x80000000 在 int 有符号类型的实际值（真值）为 -2147483648。

b = -2147483648 = 0x80000000 正好在 int 取值范围最小值，所以机器值就是真值。

c = 2147483649 = 0x80000001（机器数）> 2147483647 = 0x7fffffff，再次 overflow 越过了符号位，所以溢出后的实际值应该为负数-2147483647。

同理,unsigned short 类型的最大值是 65536, d = 65539 = 0x10011 > 65536 = 0xffff, 此时会将立即数进行位截断，只保留最后 2 个字节，即 d = 0x0011 = 3（实际值），也是在

机器中保存的机器值。

$e = -32790 = 0x7fea < -32767 = 0x8000$ ，而机器数 $0x7fea$ 对应的有符号 short 类型即 32746。

再对 warnings 分析：针对 unsigned short, 大数字隐式位截断成为 unsigned short; 针对 short 类型，在隐式类型转换中发生了溢出。这些 warnings 和刚才的分析相符合。从这里可以看出，立即数保存的时候是默认比 short 大的类型保存的，以 int 4 个字节保存，当赋值给 short 或者 unsigned short 时会发生隐式类型转换，其中就有可能因为位截断或者溢出使值与原来的值不同。

6. 完成书上第二章习题中第 40 题，提交代码，并在程序中以十六进制形式打印变量 u 的机器数。

```
#include <stdio.h>
#include <math.h>
float u2f(unsigned u)
{
    return *((float*)&u);
}

float fpower2(int x)
{
    unsigned exp, frac, u;
    if(x < -149)
    {
        exp = 0; frac = 0;
    }
    else if(x < -126)
    {
        exp = 0; frac = pow(2, x+149);
    }
    else if(x < 128)
    {
        exp = x + 127; frac = 0;
    }
    else
    {
        exp = 255; frac = 0;
    }
    u = exp << 23 | frac;
    return u2f(u);
}

int main()
{
    int x;
```



```

scanf("%d",&x);
float f = fpower2(x);
printf("0x%lX", *(unsigned long int*)&f);
return 0;
}

```

答:

本题可根据浮点数的机内表示来填空。首先当 x 指数太小时, 会发生正下溢, 此时需要填写的即 float 正数时的最小精度, 即非规格化数的最小值 $0.0...01 \times 2^{-126} = 2^{-149}$, 因为发生溢出, 所以直接填写全 0 表示 0.0; 非规格化数的精度在 $2^{-149} \sim 2^{-126}$ 之间, $x < -127$ 时返回非规格化数, 此时格式为 $0.\text{frac} \times 2^{-126} = 2^{-(n+126)} = 2^x$, $23-n=x+149$, 故 frac 需要 $2^{x+149} = \text{pow}(2, x+149)$, 此时 $\text{exp} = 0$; 规格化数的精度在 $2^{-126} \sim 2^{-127}$ 之间, $x < 128$ 时返回规格化数, 此时格式为 $1.\text{frac} \times 2^{\text{exp}-127} = 2^x$, 故 $\text{frac}=0$, $\text{exp}-127=x$, $\text{exp}=127+x$; 当值太大时, 会发生正上溢, $x > 128$ 时返回 $+\infty$, $\text{exp}=255$, $\text{frac}=0$ 。

对于 u2f 函数, 可以采用 union 的思想, 将 float 指针强制指向 unsigned int 类型的地址, 会将 unsigned int 的机器编码直接运用成 float 的机器编码, 做了强制类型转换。

对于输出 float 类型的机内编码十六进制, 同理, 可使用 `printf("0x%lX", *(unsigned long int*)&f);` 将 float 强制类型转换为 unsigned long int, 再直接使用 c 语言 printf 的输出语句 %lX 即可。

7. 编译运行以下程序, 并至少重复运行 3 次。

要求:

- (1) 给出每次运行的结果截图。
- (2) 每次运行过程中, 是否每一次循环中的判等结果都一致? 为什么?
- (3) 每次运行过程中, 每一次循环输出的 $i \quad x \quad y$ 的结果分别是什么? 为什么?

答: (1) 三次运行结果如下, 运行结果均相同: $x=32.001000$ 和 $y=31.001000$ 时结果不相等

```

chty627@ubuntu:~/Documents/homework1$ ./test1
equal
0, 22.001000, 21.001000
equal
1, 23.001000, 22.001000
equal
2, 24.001000, 23.001000
equal
3, 25.001000, 24.001000
equal
4, 26.001000, 25.001000
equal
5, 27.001000, 26.001000
equal
6, 28.001000, 27.001000
equal
7, 29.001000, 28.001000
equal
8, 30.001000, 29.001000
equal
9, 31.001000, 30.001000
equal
10, 32.001000, 31.001000
not equal
11, 33.001000, 32.001000
equal
12, 34.001000, 33.001000
equal
13, 35.001000, 34.001000
equal
14, 36.001000, 35.001000
equal
15, 37.001000, 36.001000
equal
16, 38.001000, 37.001000
equal
17, 39.001000, 38.001000
equal
18, 40.001000, 39.001000
equal
19, 41.001000, 40.001000

chty627@ubuntu:~/Documents/homework1$ ./test1
equal
0, 22.001000, 21.001000
equal
1, 23.001000, 22.001000
equal
2, 24.001000, 23.001000
equal
3, 25.001000, 24.001000
equal
4, 26.001000, 25.001000
equal
5, 27.001000, 26.001000
equal
6, 28.001000, 27.001000
equal
7, 29.001000, 28.001000
equal
8, 30.001000, 29.001000
equal
9, 31.001000, 30.001000
not equal
10, 32.001000, 31.001000
not equal
11, 33.001000, 32.001000
equal
12, 34.001000, 33.001000
equal
13, 35.001000, 34.001000
equal
14, 36.001000, 35.001000
equal
15, 37.001000, 36.001000
equal
16, 38.001000, 37.001000
equal
17, 39.001000, 38.001000
equal
18, 40.001000, 39.001000
equal
19, 41.001000, 40.001000

chty627@ubuntu:~/Documents/homework1$ ./test1
equal
0, 22.001000, 21.001000
equal
1, 23.001000, 22.001000
equal
2, 24.001000, 23.001000
equal
3, 25.001000, 24.001000
equal
4, 26.001000, 25.001000
equal
5, 27.001000, 26.001000
equal
6, 28.001000, 27.001000
equal
7, 29.001000, 28.001000
equal
8, 30.001000, 29.001000
equal
9, 31.001000, 30.001000
not equal
10, 32.001000, 31.001000
not equal
11, 33.001000, 32.001000
equal
12, 34.001000, 33.001000
equal
13, 35.001000, 34.001000
equal
14, 36.001000, 35.001000
equal
15, 37.001000, 36.001000
equal
16, 38.001000, 37.001000
equal
17, 39.001000, 38.001000
equal
18, 40.001000, 39.001000
equal
19, 41.001000, 40.001000

```

(2) 每次运行时, 不是每个结果均相等, 当 $x=32.001000$ 和 $y=31.001000$ 时, $(x-y)==1.0$ 不成立。

原因是：

浮点数的机内表示和运算的差异造成了细微的差异。

可以根据(6)中的方法输出浮点数的机内十六进制表示，先查看 x 和 y，即

```
printf("x = 0x%lx\n",*(unsigned long int*)&x);
printf("y = 0x%lx\n",*(unsigned long int*)&y);
printf("x - y = 0x%lx\n",*(unsigned long int*)&z);
printf("1.0 = 0x%lx\n",*(unsigned long int*)&a);
```

```
x = 0x40400020c49ba5e3
y = 0x403f004189374bc7
x - y = 0x3feffffffffffffe0
1.0 = 0x3ff0000000000000
```

可以得到机内表示：

可知，由于小数点 0.001 使得尾数发生了截断，并且两者的阶码正好相差 1。

浮点数进行加减操作时，需要经过对阶、尾数加减、规格化和舍入 4 个步骤。

①对阶：y 尾数右移 1，对阶后尾数左边 1 为隐含 1，右方 1 为保护位。

②尾数相减：执行 x-y 操作后

③规格化：需要进行左规，尾数向左移，阶减。

④舍入：根据 c 语言规则进行取舍

最后 x-y 得到的结果无限接近于 1.0 但是不等于 1.0，因为在上述操作中，由于 double 精度有限，保存的 0.001 并不是真实中与 0.001 相等，只能无限趋近。因为在 x-y 中产生了尾数移动相减的操作，导致这个不精确暴露了出来，使得 x-y 并不为 1.0。

其他情况中，因为他们阶相同，没有对阶使尾数移动再比较，精确度没有受到挑战。

(3) 每次输出的 i 是当前循环次数。x 与 y 进行比较后，各自加一后才进行输出！x = 22.001000 + i; y = 21.001000 + i;

三、实验源码

1.

```
#include <stdio.h>
#pragma pack(2)
struct test1
{
    char x1[17];
    short x2[2];
    int x3;
    long long x4;
};
struct test2
{
    char x1[17];
    short x2[2];
    int x3;
    long long x4;
}__attribute__((aligned(16)));

int main()
{
```



```

    test1 temp1;
    test2 temp2;
    printf("test1:\n");
    printf("size of test1= %d\n",sizeof(temp1));
    printf("positions of x1,x2,x3,x4 are: %d,%d,%d,%d\n\n",&(((test1*)0)->x1)
    ,&(((test1*)0)->x2),&(((test1*)0)->x3),&(((test1*)0)->x4));
    printf("test2:\n");
    printf("size of test2= %d\n",sizeof(temp2));
    printf("positions of x1,x2,x3,x4 are: %d,%d,%d,%d",&(((test2*)0)->x1)
    ,&(((test2*)0)->x2),&(((test2*)0)->x3),&(((test2*)0)->x4));
    //用于表示每个类型内部的起始位置
    return 0;
}

```

2.

```

#include <stdio.h>
int main()
{
    printf("test1: -2 < 2 \nanswer = %d\n",(-2<2));
    printf("test2: -2 < 2u \nanswer = %d\n",(-2<2u));
    return 0;
}

```

3.反汇编查看命令 disassemble main

4.无

5.

```

int main()
{
    int a = 2147483648;
    int b = -2147483648;
    int c = 2147483649;
    unsigned short d = 65539;
    short e = -32790;
    printf("a = %d\nb = %d\nc = %d\nd = %d\ne = %d\n",a,b,c,d,e);
    return 0;
}

```

6.见解析。

7.

```

#include <stdio.h>
#include <math.h>

```

```

int main()
{
    double x,y,z;

```

```

    x = 32.001000;
    y = 31.001000;
    z = x - y;
    printf("x = 0x%lx\n",*(unsigned long int*)&x);
    printf("y = 0x%lx\n",*(unsigned long int*)&y);
    printf("x - y = 0x%lx\n",*(unsigned long int*)&z);
    return 0;
}

```

三、实验感受

本次实验充分地体会到了计算机中表示整数、浮点数的规则，以及这些数的运算规律。了解了计算机底层工作原理，能够自我解释出一些机器运行程序与逻辑不符合现象。并且我也养成了边做实验边探索、回顾课堂知识的过程中撰写实验报告，更进一步提高了表达能力、为之后与同行交流打下了语言基础。不仅是要学会这些知识，还需要内化，成为自己的一部分，能够讲述出来这些知识，与他人沟通。

在实验过程中，也遇到了一些小困难，比如一些专有概念的不熟悉导致分析结果时出现差错，后来只好重新再把书看一遍，才能够完全理解。

总体来说这次实验耗时比我想象中的要多，因为很多概念没有理解清楚。下次在做实验和 homework 时一定要先看书和 ppt，有些纠结了好久的问题就是书上自己漏掉没有看到的关键自然段。

期待下一次与你相见。

Reference:

//reference1: __attribute(aligned(n))的解释
<https://www.cnblogs.com/ransn/p/5081198.html>
 //reference2: #pragma pack(n)的解释
<https://baike.baidu.com/item/%23pragma%20pack/3338249?fr=aladdin>
 //reference3: union 解释
<http://c.biancheng.net/view/375.html>
 //reference4: 大小端解释
https://blog.csdn.net/zhentao_chen/article/details/9247639