计算机系统基础实验报告

一、实验名称

数据的表示

二、实验目的

熟悉数字在计算机内部的表示方式，以及相关的存储、运算问题

掌握相关处理语句

三、实验环境

windows10，Visual Studio2017

Debian9，gcc 6.3.0

处理器位数：64 位

四、实验内容（结果与分析）

1.存储器在存储数据时并不是完全按照顺序的，而是采用按边界对齐的方式，这样虽然浪费了一些空间，但是减少了访问次数，权衡时间空间效率后更为合适。要计算一个结构体占据空间的大小，首先要找到元素类型中最大的一个，例如本题中 **test1** 里面最大的是 long long，8 字节，所有元素起始位置都是 8 的倍数，结构如下：

x1	x1	x1	x1	x1	x1	x1	x1
x1	x1	x1	x1	x1	x1	x1	×
x2[0]	x2[0]	x2[1]	x2[1]	x2[2]	x2[2]	×	×
x3	x3	x3	x3	×	×	×	×
x4	x4	x4	x4	x4	x4	x4	x4

（“×”代表为空）

预测该结构占用空间 40 字节，各分量相对起始位置偏移量分别是 0，16，24，32

test2 后面加上__attribute__((aligned(16)))以后该类型的变量在分配空间时，其存放的地址一定是按照 16 字节对齐（除非其中有成员大小大于 16 字节），占用空间大小也是 16 的整数倍。结构如下：

x1	x1	x1	x1	x1	x1	x1	x1	x1	x1	x1	x1	x1	x1	x1	×
x2[0]	x2[0]	x2[1]	x2[1]	x2[2]	x2[2]	×	×	×	×	×	×	x3	x3	x3	x3
x4	x4	x4	x4	x4	x4	x4	x4	×	×	×	×	×	×	×	×

预测占用空间 48 字节，各分量相对起始位置偏移量分别为 0，16，28，32.

写程序检验：因为比较熟悉所以先在 vs2017 上编写程序验证，发现输入_attribute_(()aligned(16))报错，经查是因为这个语句只适用于 linux，于是改用 debian(32bits)系统编写

```c
#include<stdio.h>
#include<stdlib.h>
#define OFFSET(type,field) ((size_t)&(((type*)0)->field))
//#pragma pack(2)
#pragma pack(32)
struct test1{
    char x1[15];
    short x2[3];
    int x3;
    long long x4;
};
struct test2{
    char x1[15];
    short x2[3];
    int x3;
    long long x4;
}__attribute__((aligned(16)));

int main()
{
    printf("struct size is:%d\n",sizeof(struct test1));
    printf("allocate x1 on address:%d\n",OFFSET(struct test1,x1));
    printf("allocate x2 on address:%d\n",OFFSET(struct test1,x2));
    printf("allocate x3 on address:%d\n",OFFSET(struct test1,x3));
    printf("allocate x4 on address:%d\n",OFFSET(struct test1,x4));

    printf("struct size is %d\n",sizeof(struct test2));
    printf("allocate x1 on address:%d\n",OFFSET(struct test2,x1));
    printf("allocate x2 on address:%d\n",OFFSET(struct test2,x2));
    printf("allocate x3 on address:%d\n",OFFSET(struct test2,x3));
    printf("allocate x4 on address:%d\n",OFFSET(struct test2,x4));
    return 0;
}
```

gcc 编译运行得到：

```
veraaaaa@debian:~/Templates$ gcc hw1_1.c -o hw1_1
veraaaaa@debian:~/Templates$ ./hw1_1
struct size is:36
allocate x1 on address:0
allocate x2 on address:16
allocate x3 on address:24
allocate x4 on address:28
struct size is 48
allocate x1 on address:0
allocate x2 on address:16
allocate x3 on address:24
allocate x4 on address:28
```

发现与自己的猜测不符：

1）可能是因为 gcc 编译器默认是以 4 字节为一组对齐，即

x1	x1	x1	x1
x1	x1	x1	x1
x1	x1	x1	x1
x1	x1	x1	×
x2[0]	x2[0]	x2[1]	x2[1]
x2[2]	x2[2]	×	×
x3	x3	x3	x3
x4	x4	x4	x4
x4	x4	x4	x4

所以该结构体占用空间为 36 字节，各分量相对起始位置偏移量分别为 0，16，24，28

2）然后 test2 中 x3 偏移量不同可能是因为紧跟 x2 后面，没有靠右。

若使用**#pragma pack(2)伪指令**，则对于大小大于等于 2 的成员变量按照 2 对齐，否则按照默认的 4 字节对齐，与之前不同之处就在于 x3 不需要因为另起一行前面空两个字节，所以偏移量减少 2，而 test1 的总空间减少 2，test2 因为最终需要是 16 的倍数所以总大小必须扩展到 48。

```
veraaaaa@debian:~/Templates$ gcc hw1_1.c -o hw1_1
veraaaaa@debian:~/Templates$ ./hw1_1
struct size is:34
allocate x1 on address:0
allocate x2 on address:16
allocate x3 on address:22
allocate x4 on address:26
struct size is 48
allocate x1 on address:0
allocate x2 on address:16
allocate x3 on address:22
allocate x4 on address:26
```

结构如图：

x1	x1
x1	x1
x1	x1
x1	x1
x1	x1
x1	x1
x1	x1
x1	×
x2[0]	x2[0]
x2[1]	x2[1]
x2[2]	x2[2]
x3	x3
x3	x3
x4	x4
x4	x4
x4	x4
x4	x4

若使用**#pragma pack(32)伪指令**，因为该结构体中所有成员大小都小于 32，所以按照本身的对齐方式排列，与第一次的结果一样

```
veraaaaa@debian:~/Templates$ gcc hw1_1.c -o hw1_1
hw1_1.c:5:9: warning: alignment must be a small power of two, not 32 [-Wpragmas]
 #pragma pack(32)
         ^~~~

veraaaaa@debian:~/Templates$ ./hw1_1
struct size is:36
allocate x1 on address:0
allocate x2 on address:16
allocate x3 on address:24
allocate x4 on address:28
struct size is 48
allocate x1 on address:0
allocate x2 on address:16
allocate x3 on address:24
allocate x4 on address:28
```

2. -1 < 1 和 -1 < 1u 结果不一样

因为-1<1 是按照有符号数的大小比较，-1-1=-1+(-1)=11…1B+11…1B=11…10，最高位是符号位，是负数，所以 11…1B(-1)<00…1B(1)，正确，结果为 1；

而-1<1u，int 和 unsigned int 类型同时比较时，按照无符号数比较，即 11…1B 与 00…1B 的比较，此时最高位不是符号位显然前者大于后者，所以结果为 0。

gcc 编译运行结果符合解释。

```c
#include<stdio.h>
int main()
{
    int a=(-1<1)?1:0;
    int b=(-1<1u)?1:0;
    printf("the result of -1<1 is %d\n",a);
    printf("the result of -1<1u is %d\n",b);
    return 0;
}
```

```
veraaaaa@debian:~/Templates$ gcc hw1_2.c -o hw1_2
veraaaaa@debian:~/Templates$ ./hw1_2
the result of -1<1 is 1
the result of -1<1u is 0
```

3.运行程序结果：

```
veraaaaa@debian:~/Templates$ gcc hw1_3.c -o hw1_3
veraaaaa@debian:~/Templates$ ./hw1_3
unsigned int is 0
unsigned short is 1
```

数据在比较大小时存在数据类型提升的概念，有两种情况：

1） 符号扩展：对于带符号数，新的高位使用当前最高位即符号位进行填充

2） 零扩展：对于无符号数，在新的最高位直接填 0

因为所有比 int 型小的数据类型（包括 char, short）转换成 int 型时，如果转换后的数据超出 int 型所能表示的范围，则转换成 unsigned int 型。

题中 c 和 a 在比较时 c 先转换成 int 型，再转换成 unsigned 型：11…1B（有符号数按照最高位进行扩位），而此时 a 的大小为 1B，无符号数比较显然 c 大于 a，所以第一次输出结果是 unsigned int is 0。

当 b 和 c 比较时，c 应该转换成 int 型，为 11…1B，有符号数最高位表示符号，所以是-(2^31-1)=-21473647。b 也应该转换成 int 型，为 00…01B，有符号数比较，显然 b>c，所以输出结果为 unsigned short is 1.

4.运行结果是

```
veraaaaa@debian:~/Templates$ gcc hw1_4.c -o hw1_4
veraaaaa@debian:~/Templates$ ./hw1_4
0x43
0xffffff87
0x21
0x65
```

由于在 union 共同体中，int a 和 char b[4]数据公用同一段内存地址，而此时 a 和 b 同时占用四个字节。当执行 num.a=0x87654321 时，b 也会被赋值，内存示意图如下：

b 的值　　8　7　6　5　4　3　2　1

b 的地址　　b[3]　　b[2]　　b[1]　　b[0]

所以最后输出 b[1]和 b[3]会输出 0x43 和 0xffffff87（因为是最高位）

这也表明 debian9 是小端存储方式，即数组中元素下标越小的地址越小，下标越大的地址越大。验证其他两个元素：b[0]=0x21; b[2]=65。

5.机器数就是数值在计算机中的表示形式，真值则是它在现实中的实际数值。机器数的主要特点在于符号的数字化和数的位数受机器字长的限制。
a 的机器数是 10…0B
b 的机器数是 10…0B
c 的机器数是 10…1B
d 的机器数无法表示
e 的机器数也无法表示
int 类型的取值范围是-21473648~+21473647, b 是正确的, 而 a 其实已经超出了表示范围，但是因为 a=21473648=21473647+1=01…1B+1B=10…0B=-21473648，而-21473648 在 int 型的表示范围内，所以不会发生溢出。如果输出 a 的十进制数就得到-21473648。同理 c=21473649=-21473647，因此也不会发生溢出
d 是无符号整型数，只有 16 位，表示范围是 0-2^16-1（0-65535），65539 超出了这个范围，所以会有溢出 warning
e 是有符号的短整型变量，表示范围是-2^15~2^16-1（-32768~+32767），-32790=-32768-22=-32768+（-22）=0001 0111 1…10 1001B，是一个很大的正数，远超表示范围，所以会有溢出 warning
debian 上编写程序进行检验：

```c
#include<stdio.h>
int main()
{
    int a=2147483648;
    int b=-2147483648;
    int c=2147483649;
    unsigned short d=65539;
    short e=-32790;
    printf("a=%x\n",a);
    printf("b=%x\n",b);
    printf("c=%x\n",c);
    printf("d=%x\n",d);
    printf("e=%x\n",e);
    return 0;
}
```

输出警告信息为：

```
veraaaaa@debian:~/Templates$ gcc hw1_5.c -o hw1_5
hw1_5.c: In function ▪main▪:
hw1_5.c:7:19: warning: large integer implicitly truncated to unsigned type [-Wov
erflow]
  unsigned short d=65539;
                   ^~~~~

hw1_5.c:8:10: warning: overflow in implicit constant conversion [-Woverflow]
  short e=-32790;
          ^
```

运行结果：

```
veraaaaa@debian:~/Templates$ ./hw1_5
a=80000000
b=80000000
c=80000001
d=3
e=7fea
```

这道题收获很大：

1） 重新理解了带符号数和无符号数溢出的区别

无符号数的溢出比较简单，是指在有限的位数中无法表示该数，即 n 位数超出了-2^n~2^n-1 的范围；

而带符号数的溢出是指在表示数值的 n-1 位无法表示该数的值时，占用了最高位（符号位），可能会导致符号的改变等，但是带符号数的这种溢出只是得到了错误的值，但并没有引发错误（overflow）。

2） VS 对-2147483648 报错的原因：

在 VS2017 上运行该程序时，总是报错：一元负运算应用于无符号类型，结果仍为无符号类型。经了解是因为编译器在看到 int x=-2147483648 时，首先判断 21473648>INT_MAX，知道 int 装不下，决定使用 unsigned int，发现前面还有一个负号，于是对 2147483648 取反。而这个数取反后仍为它本身，造成隐患，所以出现 error。解决办法是：定义 INT_MIN，值为-2147483647-1.

6. IEEE754 标准中 32 位单精度浮点数的格式为 1 位符号位 S+8 位阶码 E+23 位尾数 F，偏置常数为 127。由课本可以知道单精度浮点数所能表示的所有整数的临界点

E	F	加上偏置常数	描述
0000 0000	0000 0000 0000 0000	-126	零
0000 0000	0000 0000 0000 0001	-126	最小非规格化数
0000 0000	1111 1111 1111 1111	-126	最大非规格化数
0000 0001	0000 0000 0000 0000	-126+127=1	最小规格化数
1111 1110	1111 1111 1111 1111	127+127=254	最大规格化数
1111 1111	0000 0000 0000 0000	128+127=255	无穷大

所以 2^x 各种浮点数表示形式的临界值有：（-∞,-150],[-149,-127],[-126,127],[128,+∞）

检验程序如下：

其中 u2f 函数用于把无符号整型数转化成 float 型，fpower2 实现函数功能，输出 2^x 的十进制和十六进制形式。

```
#include<stdio.h>
float u2f(unsigned u)
{
    return *(float *)&u;
}
float fpower2(int x)
{
    unsigned exp,frac,u;
    if(x<-149)
    {
        //the value is too small, return 0.0
        exp=0;
        frac=0;
    }
    else if(x<-126)
    {
        //return denormalized number
        exp=0;
        frac=1<<(x+149);
    }
    else if(x<128)
    {
        //return normalized number
```

```
24          exp=x+127;
25          frac=0;
26      }
27      else
28      {
29          //the value is too big, return infinite
30          exp=255;
31          frac=0;
32      }
33      u=exp<<23|frac;
34      printf("2^%d=%fD=0x%x\n",x,u2f(u),u);
35      return u2f(u);
36  }
37  int main()
38  {
39      int x;
40      printf("Please input an integer: \n");
41      scanf("%d",&x);
42      fpower2(x);
43      return 0;
44  }
```

运行结果：以 2 的二次方、五次方、十次方为例

```
veraaaaa@debian:~/Templates$ gcc hw1_6.c -o hw1_6
veraaaaa@debian:~/Templates$ ./hw1_6
Please input an integer:
2
2^2=4.000000D=0x40800000
veraaaaa@debian:~/Templates$ ./hw1_6
Please input an integer:
5
2^5=32.000000D=0x42000000
veraaaaa@debian:~/Templates$ ./hw1_6
Please input an integer:
10
2^10=1024.000000D=0x44800000
```

7.（1）三次运行结果

```
veraaaaa@debian:~/Templates$ gcc hw1_7.c -o hw1_7
veraaaaa@debian:~/Templates$ ./hw1_7
not equal
#0:2.001000,1.001000
equal
#1:3.001000,2.001000
equal
#2:4.001000,3.001000
not equal
#3:5.001000,4.001000
equal
#4:6.001000,5.001000
equal
#5:7.001000,6.001000
equal
#6:8.001000,7.001000
equal
#7:9.001000,8.001000
equal
#8:10.001000,9.001000
equal
#9:11.001000,10.001000
```

```
veraaaaa@debian:~/Templates$ gcc hw1_7.c -o hw1_7
veraaaaa@debian:~/Templates$ ./hw1_7
not equal
#0:2.001000,1.001000
equal
#1:3.001000,2.001000
equal
#2:4.001000,3.001000
not equal
#3:5.001000,4.001000
equal
#4:6.001000,5.001000
equal
#5:7.001000,6.001000
equal
#6:8.001000,7.001000
equal
#7:9.001000,8.001000
equal
#8:10.001000,9.001000
equal
#9:11.001000,10.001000
```

```
veraaaaa@debian:~/Templates$ gcc hw1_7.c -o hw1_7
veraaaaa@debian:~/Templates$ ./hw1_7
not equal
#0:2.001000,1.001000
equal
#1:3.001000,2.001000
equal
#2:4.001000,3.001000
not equal
#3:5.001000,4.001000
equal
#4:6.001000,5.001000
equal
#5:7.001000,6.001000
equal
#6:8.001000,7.001000
equal
#7:9.001000,8.001000
equal
#8:10.001000,9.001000
equal
#9:11.001000,10.001000
```

（2）不是每次判等结果都一致。

因为浮点数的加减运算比较复杂，需要对阶并且最终进行舍入，因此会导致“大数吃小数”的亲恩替，使其不满足加法结合律。例如 1.001 和 0.001 做减法时，因为对阶使得小数 0.001 尾数中的有效数字右移后被丢弃，使得小数变为 0 导致 x-y≠z。因为转化成二进制后 1/0 位置不确定，所以会出现有时相等有时不等的情况。

（3）每次循环输出的 i、x、y 结果如图，随着 i 递增，x，y 也依次递增。

因为加上 1.0 不会产生因为移位造成的数据丢失。

8. 32 位 debian 运行程序

```c
#include<stdio.h>
int main()
{
    int x=-1;
    unsigned u=2147483648;
    printf("x=%u=%d\n",x,x);
    printf("u=%u=%d\n",u,u);
    return 0;
}
```

```
veraaaaa@debian:~/Templates$ gcc hw1_8.c -o hw1_8
veraaaaa@debian:~/Templates$ ./hw1_8
x=4294967295=-1
u=2147483648=-2147483648
```

分析原因：-1 的机器码是 00…01B 的补码 11…1B，在作为无符号数输出时，首位不再是符号位，所以该数表示 2^32-1=4294967295，真值就是-1。
而 2147483648=2^31，机器码是 10…0B，以无符号数形式输出时，就表示 2^31，即 2147483648，以%d 形式输出时，则是一个负数，即-2^31，即-2147483648

9.

1） 在 ISO C90 标准下，2147483648 为 unsigned int 型，因此-2147483648<2147483647 按照无符号数比较，10…0B 比 01…1B 大，所以结果为 false
而在 ISO C99 标准下，2147483648 为 long long 型，因此按照带符号数比较，此时 10…0B 是一个负数，比 01…1 要小，所以结果为 true.

2） i<2147483647，i 是 int 型数，按照带符号的 int 型数比较，由 1 可知结果为 true

3） -2147483647-1<2147483647 也按照 int 型数比较，结果仍未 true

```c
#include<stdio.h>
int main()
{
    int a=(-2147483648<2147483647)?1:0;
    printf("the result of -2147483648<2147483647 is:%d\n",a);

    int i=-2147483648;
    int b=(i<2147483647)?1:0;
    printf("the result of i<2147483647 is:%d\n",b);

    int c=(-2147483647-1<2147483647)?1:0;
    printf("the result of -2147483647-1<2147483647 is:%d\n",c);
    return 0;
}
```

```
veraaaaa@debian:~/Templates$ gcc hw1_9.c -o hw1_9
veraaaaa@debian:~/Templates$ ./hw1_9
the result of -2147483648<2147483647 is:1
the result of i<2147483647 is:1
the result of -2147483647-1<2147483647 is:1
```

发现自己的 gcc 用的是 C99 标准，上网查到可以用-std=c90 实现转换到 C90 标准

```
veraaaaa@debian:~/Templates$ gcc -std=c90 hw1_9.c -o hw1_9
hw1_9.c: In function ‘main’:
hw1_9.c:4:2: warning: this decimal constant is unsigned only in ISO C90
  int a=(-2147483648<2147483647)?1:0;
  ^~~
hw1_9.c:7:2: warning: this decimal constant is unsigned only in ISO C90
  int i=-2147483648;
  ^~~
veraaaaa@debian:~/Templates$ ./hw1_9
the result of -2147483648<2147483647 is:0
the result of i<2147483647 is:1
the result of -2147483647-1<2147483647 is:1
```

10.在 C90 和 C99 标准下运行结果部分相同，部分不相同。
C90 标准

```
veraaaaa@debian:~/Templates$ gcc -std=c90 hw1_10.c -o hw1_10
hw1_10.c: In function ‘main’:
hw1_10.c:5:2: warning: this decimal constant is unsigned only in ISO C90
  unsigned u=2147483648;
  ^~~~~~~~
hw1_10.c:9:2: warning: this decimal constant is unsigned only in ISO C90
  if(-2147483648<2147483647)
  ^~
hw1_10.c:20:2: warning: this decimal constant is unsigned only in ISO C90
  if(-2147483648-1<2147483647)
  ^~
hw1_10.c:25:2: warning: this decimal constant is unsigned only in ISO C90
  else if(-2147483648-1==2147483647)
  ^~~~
veraaaaa@debian:~/Templates$ ./hw1_10
x=4294967295=-1
u=2147483648=-2147483648
-2147483648<2147483647 is false
-2147483648-1==2147483647
```

C99 标准

```
veraaaaa@debian:~/Templates$ gcc hw1_10.c -o hw1_10
veraaaaa@debian:~/Templates$ ./hw1_10
x=4294967295=-1
u=2147483648=-2147483648
-2147483648<2147483647 is true
-2147483648-1<2147483647
```

原因：

首先说明两个标准关于本题的区别在于对 2147483648 的处理，即在 ISO C90 标准下，认为是 unsigned int 型，而在 ISO C99 标准下，认为是 long long 型

1）-1 的二进制补码表示为 11…1B，2147483648 的二进制表示为 10…0u，且程序中已经说明是 unsigned 型，所以两个输出没有区别。

2）由上面第 9 题第 1）问可知 C90 下-2147483648<2147483647 为假，C99 下为真

3）C90：-2147482638-1=10…00-00…01=01…1，2147483647=01…1，

按照无符号整型比较，前者等于后者

C99：-2147483648-1=11…10…0+11…11…1=11…01…1，2147483647=00…01…1

按照带符号的 long long 型比较，前者是负数，后者是正数，所以是小于。

五、实验体会

通过这次实验（作业），我对数据在计算机中如何表示，如何存储和运算有了更深的了解：明白了结构体中成员的对齐理论，知道了在空间和时间产生冲突时，一般会权衡之后选择时间更短；联合这一特殊的形式使得多个成员“共享”一段空间；无符号数和带符号数、机器数和真值；不同类型数据之间（尤其是浮点数）的大小比较、类型相互转换、加减乘除；也了解了 IEEE754 标准，C90/C99 的差异，感受到了计算机世界严谨的逻辑。除了对本章内容掌握更好以外，我还加强了自己对于虚拟机的操作水平，尤其是在新的编译环境下写代码的能力。

这次实验完成工程中经常会碰到一些问题，有时是基础知识掌握不牢，有时是键入出现错误（比如 include 忘记写#），不过这些问题在多方查找学习、吸收经验教训的基础上大都获得了解决。目前剩余的问题就是需要下载 gcc 4.7/4.8 版本（开始做实验时才意识到新版与旧版的不同），然后配置好 ubuntu（使操作更简单易行）。