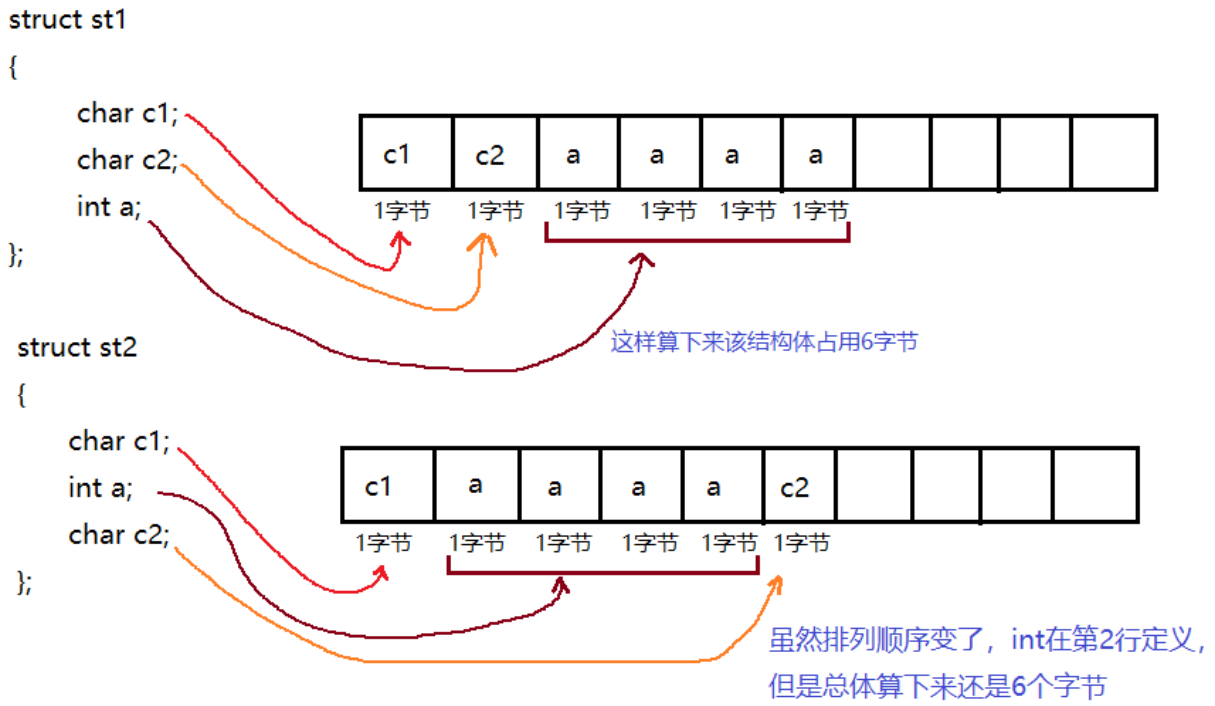
C 语言语法高级应用第 2 版

作者：向仔州

目录

结构体字节对齐问题




```c
struct st1
{
        char c1;
        char c2;
        int a;
};

struct st2
{
        char c1;
        int a;
        char c2;
};


int main(void)
{
        printf("%d %d\n",sizeof(struct st1),sizeof(struct st2));
        return 0;
}
```

```
8 12
```

两个结构体大小不一致

不对啊!! 都是 6 个字节的变量，怎么一个结构体是 8 字节，一个结构体 12 字节呢?

难道是结构体内部变量没有按照顺序对齐的原因?

其实影响结构体大小的因素，1.字节对齐(是按照 1 字节，4 字节，8 字节还是 16 字节对齐)?

2.结构体成员变量大小也有关系

3.不同类型的变量对齐

因为 GCC 编译器默认是 4 字节对齐，那么 int 满足 4 字节，但是 int 前面的 char 是 1 字节，而且地址是 4 字节一排，4 字节一排，第 1 排被两个 char 用了，那么就只有让第 2 排承载 int，所以第一排浪费了两个字节空出来。结果就是 8 字节。

```
struct st1
{
        char c1;
        char c2;
        int a;
};
```

地址	变量	说明
1000	char c1	char 1字节
1001	char c2	char 1字节
1002		
1003		
1004	int a	int 4字节，地址必须是4的倍数
1005		
1006		
1007		

为什么这两行不能给int a？ 这是因为int必须是连续的4地址才行，然后地址也4字节一个间隔，所以int只有给新的4字节地址才行

注意，结构体里面变量定义的先后顺序也是有关系的，请看下面

```
struct st2
{
        char c1;
        int a;
        char c2;
};
```

地址	变量	说明
1000	char c1	char 1字节
1001		
1002		
1003		
1004	int a	int 4字节，地址必须是4的倍数
1005		
1006		
1007		
1008	char c2	char 1字节
1009		
1010		
1011		

前面说过了，GCC编译器要求4字节对齐，那么你在结构体分配变量的时候要不然一次性挨着分配4个字节的char，或者挨着分配两个short，这种一个char中间又隔离出一个int，那么第1个char就只有占4字节地址，但是char本身只使用1字节空间

所以这个结构体大小占用 12 字节。

```c
struct st4
{
        char array[5];
        short c;
        int b;
};


int main(void)
{
        printf("%d\n",sizeof(struct st4));
        return 0;
}
```

```
12
```

这个应该是 11 字节，怎么算出 12 字节的?

```
struct st4
{
    char array[5];
    short c;
    int b;
};
```

地址	变量	说明
1000	char arr[0]	char 1字节
1001	char arr[1]	char 1字节
1002	char arr[2]	char 1字节
1003	char arr[3]	char 1字节
1004	char arr[4]	char 1字节
1005		
1006	short c	2字节
1007		
1008	int b	4字节
1009		
1010		
1011		

这就是我说的 char一次性挨着分配很多个，如果在向下分配过程中遇见short，正好char没有分配完多余的4字节地址，那么就要空出来，让short对齐，这里空出一格就是让4字节的最后两个地址给short 对齐

#pragma pack(n) 内存对齐函数

n = 1 表示按照 1 字节对齐，n=2 表示按照 2 字节对齐，这样对齐方式是紧密排列了

```c
#pragma pack(1)
struct st4
{
        char array[5];
        short c;
        int b;
};


int main(void)
{
        printf("%d\n",sizeof(struct st4));
        return 0;
}
```

Pragma 必须放在结构体前面，才能让下面结构体按照设置的对齐方式生效

```
11
```

你看计算出来只有 11 字节

结构体按照1字节对齐排列

```c
pragma pack(1)
struct st4
{
    char array[5];
    short c;
    int b;
};
```

地址	变量	说明
1000	char arr[0]	char 1字节
1001	char arr[1]	char 1字节
1002	char arr[2]	char 1字节
1003	char arr[3]	char 1字节
1004	char arr[4]	char 1字节
1005	short c	2字节
1006		
1007	int b	4字节
1008		
1009		
1010		
1011		

```c
#pragma pack(4)
struct st4
{
        char array[5];
        short c;
        int b;
};


int main(void)
{
        printf("%d\n",sizeof(struct st4));
        return 0;
}
```

4 字节对齐，又回到 GCC 编译器默认的情况了

```
12
```

地址	变量	说明
1000	char arr[0]	char 1字节
1001	char arr[1]	char 1字节
1002	char arr[2]	char 1字节
1003	char arr[3]	char 1字节
1004	char arr[4]	char 1字节
1005		
1006	short c	2字节
1007		
1008	int b	4字节
1009		
1010		
1011		

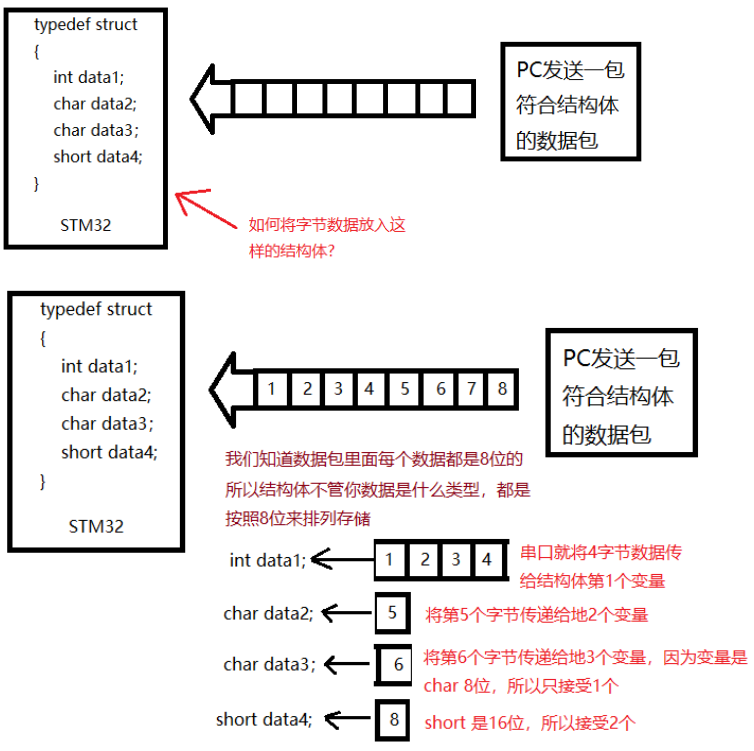
一般不建议去修改 pragma 对齐方式，因为如果你的程序要直接用在其它系统或者电脑平台上运行，可能会出现一些问题，建议就用 GCC 默认对齐方式。

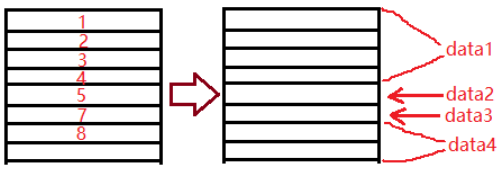
```
#pragma pack(pop) //pack(pop) 就是让以下结构体恢复到系统默认对应方式
```

```
#pragma  pack(4)  // CC 结构体 4 字节对齐
struct CC {
    double d;
    char b;
    int a;
    short c;
};
#pragma pack() //BB 结构体又恢复到系统默认对齐方式
struct BB{
    double d;
    char b;
    int a;
    short c;
};
```

用结构体方式接受串口的数据包，结构体对数据内容进行了取名，方便使用

需要上一节#pragma pack 字节对齐的知识





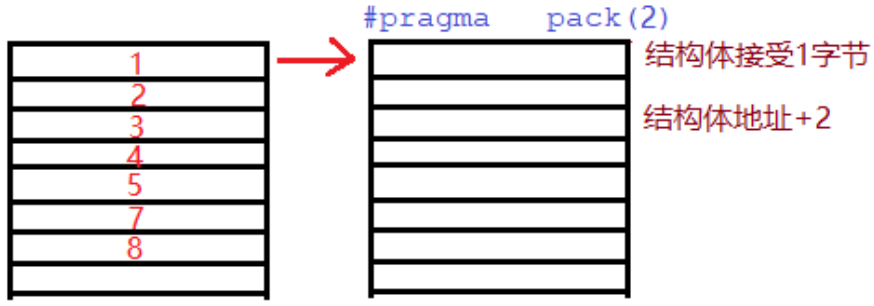
串口数据放在数组buff[8]内存布局　　结构体内存布局

将数组内存排列的数据拷贝给结构体

(注意结构体必须改成1字节对齐)

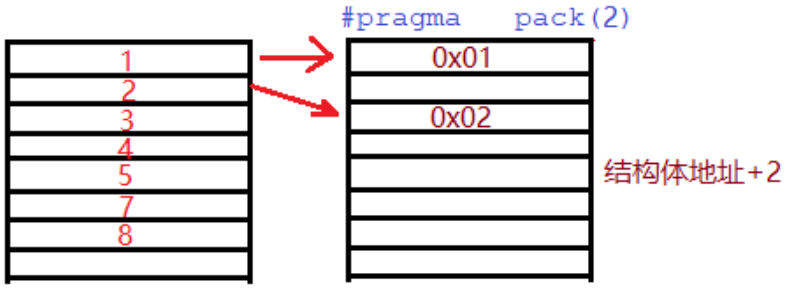
为什么结构体要改成 1 字节对齐?

因为串口是单字节数组，所以串口数据拷贝给结构体只能 1 字节 1 字节拷贝，如果结构体是 2 字节对齐，那么结构体收到串口拷贝的 1 个字节之后，结构体地址+2



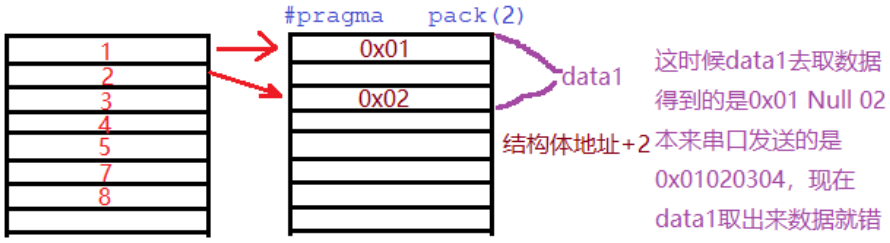
串口数据放在数组buff[8]内存布局　　结构体内存布局

现在结构体默认是2字节对齐



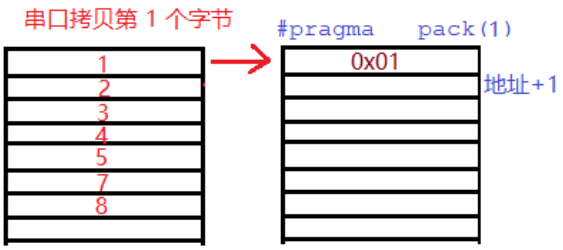
串口数据放在数组buff[8]内存布局　　结构体内存布局

串口再拷贝一个数据给结构体

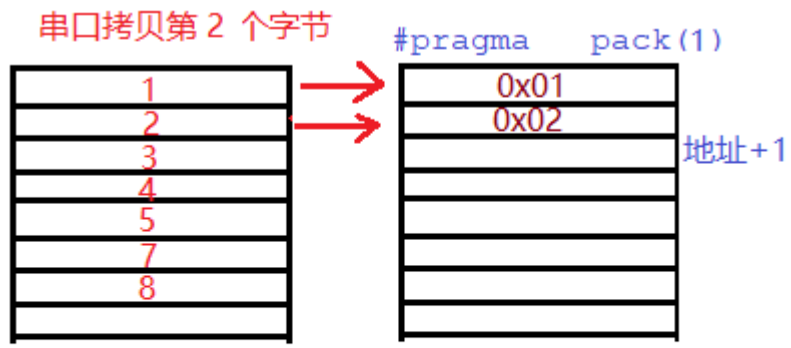


串口数据放在数组buff[8]内存布局　　结构体内存布局

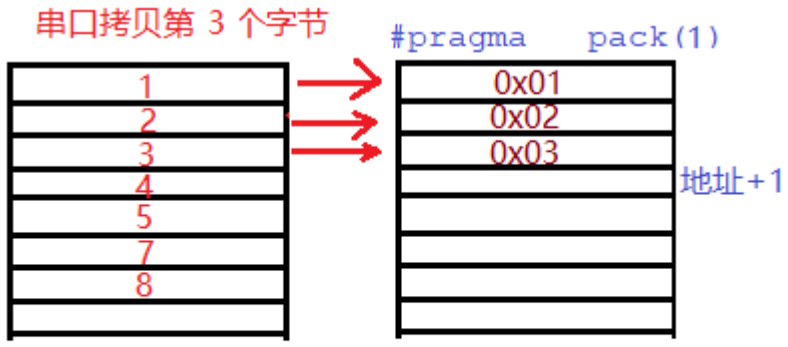
这就是结构体地址 2 字节对齐的问题



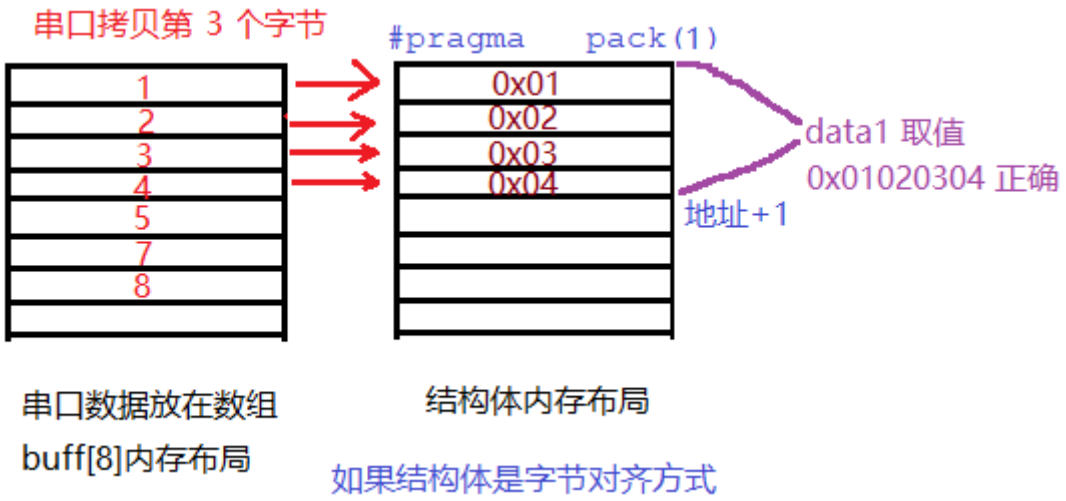
如果结构体是字节对齐方式



如果结构体是字节对齐方式



如果结构体是字节对齐方式



这就是为什么要单字节对齐的原因，因为串口拷贝给结构体的数据是按照单字节拷贝的。

```
#pragma pack(1) //以下结构体按照 1 字节对齐
typedef struct
{
	uint16_t Preamble;	//前导码
	uint32_t DeviceAddr;	//得到本设备地址
	uint8_t  FunCode;	//功能码
	uint16_t StartAddr;	//起始地址
	uint16_t DataNum;	//数据个数
	uint16_t CRC16;	//CRC 校验

}ModbusMaster_Rcv_Data; //接受串口的数据打印出来
```

ModbusMaster_Rcv_Data MBrcvdata; 创建一个结构体存放串口拷贝的数据

uint8_t data[20] = {0xCE,0xA0,0x1f,0x2f,0x3f,0x4f,0x55,0x11,0x12,0x30,0x30,0x22,0xff};

如果这一帧 data 是串口发过来的数据

memcpy(&MBrcvdata,data,20); //将串口缓存的数据拷贝给结构体

直接打印结构体里面的数据，里面的数据就是串口每一段数据的标识

printf("Preamble1 = %x\r\n", MBrcvdata.Preamble);

printf("Preamble1 = %x\r\n", MBrcvdata. DeviceAddr);

printf("Preamble1 = %x\r\n", MBrcvdata. StartAddr);

打印结果

```
Preamble1 = a0ce
DeviceAddr1 = 4f3f2f1f
StartAddr1 = 1211
```

我们发现数据大小端是反的 0xce 是低位，0xa0 是高位，貌似根据数组排列这样是正确的。

但是结构体输出也必须和数组排列一致才对，比如 Preamble1 输出应该是 0xcea0，因为数组是 data[0]=0xce，data[1] = 0xa0

所以我们需要做大小端反转

```
/*uint16_t 数据大端转小端*/
void BigLittle_endian16(uint16_t *StuData)
{
	uint16_t ret = 0;
	uint16_t tmpL;
	uint16_t tmpH;

	ret = *StuData;
	tmpL = (ret >>8 ) & 0x00ff;
	tmpH = (ret << 8 ) & 0xFF00;
```

```
    *StuData = tmpH | tmpL;
}

/*uint32_t 数据大端转小端*/
void BigLittle_endian32(uint32_t *StuData)
{
    uint32_t ret = 0;
    uint32_t tmpL1;
    uint32_t tmpL2;
    uint32_t tmpH1;
    uint32_t tmpH2;
    /*4f 3f 2f 1f*/
    /*1f 2f 3f 4f*/
    ret = *StuData;

    tmpL1 = ( ret >> 24 ) & 0x000000ff;
    tmpL2 = ( ret >> 8 ) & 0x0000ff00;
    tmpH1 = ( ret << 8 ) & 0x00ff0000;
    tmpH2 = ( ret << 24) & 0xff000000;

    *StuData = tmpH2 | tmpH1 | tmpL2 | tmpL1 ;

}
```

使用以上函数进行大小端转换，转换 16 位，和 32 位两种，8 位不需要转。

直接传入 1 字节对齐的结构体可以转换吗？BigLittle_endian16(&MBrcvdata.Preamble);

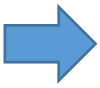
记住这样绝对不行，会出现程序宕机

因为 BigLittle_endian16 都是进行 16 位(2 字节)变量运算的

所以只有把 MBrcvdata 结构体的数据赋值给编译器默认的字节对齐结构体变量。

所以我要创建一个默认的字节对齐，相同的结构体变量。

```
#pragma pack() //使用 pack()  将以下结构体变成编译器默认的结构体对齐方式
typedef struct
{
    uint16_t Preamble;      //前导码
    uint32_t DeviceAddr;    //得到本设备地址
    uint8_t   FunCode;              //功能码
    uint16_t StartAddr;     //起始地址
    uint16_t DataNum;                   //数据个数
    uint16_t CRC16;                 //CRC 校验

}ModbusMaster_Rcv_cache;
```



```
#pragma pack(1) //以下结构体按照 1 字节对齐
typedef struct
{
    uint16_t Preamble;      //前导码
    uint32_t DeviceAddr;    //得到本设备地址
    uint8_t   FunCode;      //功能码
    uint16_t StartAddr;     //起始地址
    uint16_t DataNum;    //数据个数
    uint16_t CRC16;        //CRC 校验
}ModbusMaster_Rcv_Data;
```

因为 Rcv_cache 是系统默认对齐方式，所以将 1 字节对齐的结构拷贝过来

```
ModbusMaster_Rcv_Data     MBrcvdata;     //1 字节对齐
ModbusMaster_Rcv_cache     cache;        //系统默认对齐方式

/*将 1 字节对齐的数据拷贝给默认对齐的结构体*/
cache.Preamble = MBrcvdata.Preamble;
cache.DeviceAddr = MBrcvdata.DeviceAddr;
```

```
cache.FunCode = MBrcvdata.FunCode;
cache.StartAddr = MBrcvdata.StartAddr;

/*把默认对齐的结构体放入大小端转换程序就可以了*/
printf("Preamble1 = %x\r\n",cache.Preamble);
BigLittle_endian16(&cache.Preamble);//16 位大小端转换
printf("Preamble2 = %x\r\n",cache.Preamble);

printf("DeviceAddr1 = %x\r\n",cache.DeviceAddr);
BigLittle_endian32(&cache.DeviceAddr);
printf("DeviceAddr2 = %x\r\n",cache.DeviceAddr);

printf("FunCode = %x\r\n",cache.FunCode);

printf("StartAddr1 = %x\r\n",cache.StartAddr);
BigLittle_endian16(&cache.StartAddr);//16 位大小端转换
printf("StartAddr2 = %x\r\n",cache.StartAddr);
```

```
Preamble1 = a0ce
Preamble2 = cea0
DeviceAddr1 = 4f3f2f1f
ret 1329540895
DeviceAddr2 = 1f2f3f4f
FunCode = 55
StartAddr1 = 1211
StartAddr2 = 1112
```

大小端就转换过来了，a0ce 转换成 cea0 和数组对上了。其它也一样

__attribute__ 用法

C 语言中__attribute__((at())绝对定位的应用

__attribute__(at(绝对地址))的作用分两个，一个是绝对定位到 Flash，另个一是绝对定位到 RAM。

定位到 flash 中，一般用于固化的信息，如出厂设置的参数，上位机配置的参数，ID 卡的 ID 号，flash 标记等等

```
const u16 gFlashDefValue[512] __attribute__((at(0x0800F000))) = {0x1111,0x1111,0x1111,0x0111,0x0111,0x0111};//定位在 flash 中,其他 flash 补充为 00
const u16 gflashdata__attribute__((at(0x0800F000))) = 0xFFFF;//地址为 0x0800F000
```

定位到 RAM 中，一般用于数据量比较大的缓存，如串口的接收缓存，再就是某个位置的特定变量

```
u8 USART2_RX_BUF[USART2_REC_LEN] __attribute__ ((at(0X20001000)));//接收缓冲,最大 USART_REC_LEN 个字节,起始地址为 0X20001000.
```

注意:

绝对定位不能在函数中定义，局部变量是定义在栈区的，栈区由 MDK 自动分配、释放，不能定义为绝对地址，只能放在函数外定义

定义的长度不能超过栈或 Flash 的大小，否则，造成栈、Flash 溢出

__attribute__((__aligned__(n)))对变量的影响

__attribute__（(aligned(n)))和 pragma 效果一样用于字节对齐，

__attribute__（(aligned(n)))和 pragma 有什么区别呢？ n=1 ，1 字节对齐，n=4, 4 字节对齐 .....

```c
struct stu
{
        char sex;
        int length;
        char name[10];

};



int main(void)
{
        printf("%d\n",sizeof(struct stu));
        return 0;
}
```

```
20
```

20 个字节，理论上只有 15 个

C 语言 ，变量位域定义，如 unsigned char ch: 6; 这种用法

```c
struct bs
{
        unsigned m;
        unsigned n: 4; //存放不长度超过4位bit的数据
        unsigned c: 6; //存放不长度超过6位bit的数据

};



int main(void)
{
        struct bs domain;
        domain.m = 0xad;
        domain.n = 0xad;
        domain.c = 0xad;

        printf("m = %x \n",domain.m);
        printf("n = %x \n",domain.n);
        printf("c = %x \n",domain.c);
        return 0;
}
```

```
m = ad
n = d
c = 2d
```

你看，输出结果，m 是完整正确的，n 和 c 变量都被截取了丢掉了部分。

这是为什么呢?

这就是位域运算，给某个变量限制存放数据的范围。感觉是不是有点多此一举了?

```
struct bs
{
        unsigned m;
        unsigned n: 4; //存放不长度超过4位bit的数据
        unsigned c: 6; //存放不长度超过6位bit的数据

};
```

在这个结构体中：

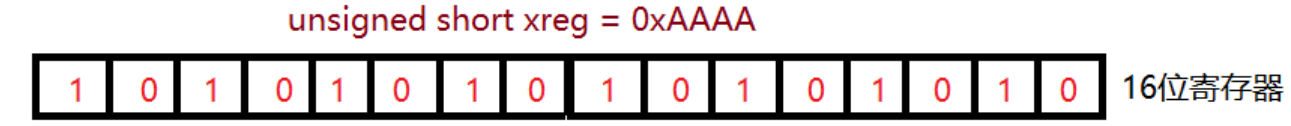
m 可以存放很大的数据

但是 n 只能存放不超过 4 位的数据，也就是最大二进制为 1111(0xf)，所以`n = d` n 变量的 a 值被截取掉了，很正常。

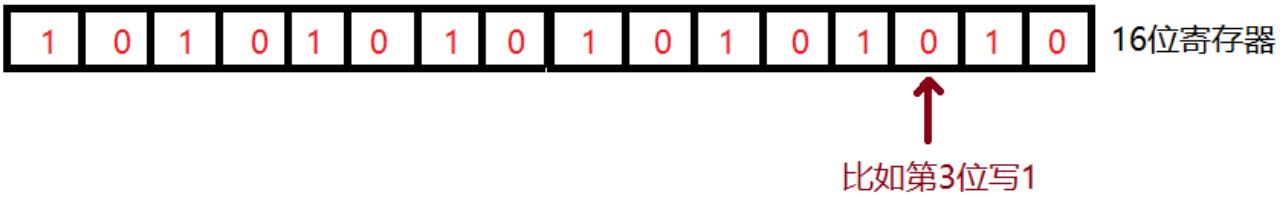
同样 c 只能存放最大不超过 6 位的数据，也就是最大二进制位 111111(0x3f)，所以 ad (1010 1101) 截取高两位就是(10 1101) 2d，`c = 2d`。

这种位域运算这么折腾到底有什么用?

我用一个 16 位的 unsigned short 变量来模拟 16 位寄存器



如果我想修改其中某一位，不使用移位运算，有更好的修改方式吗?



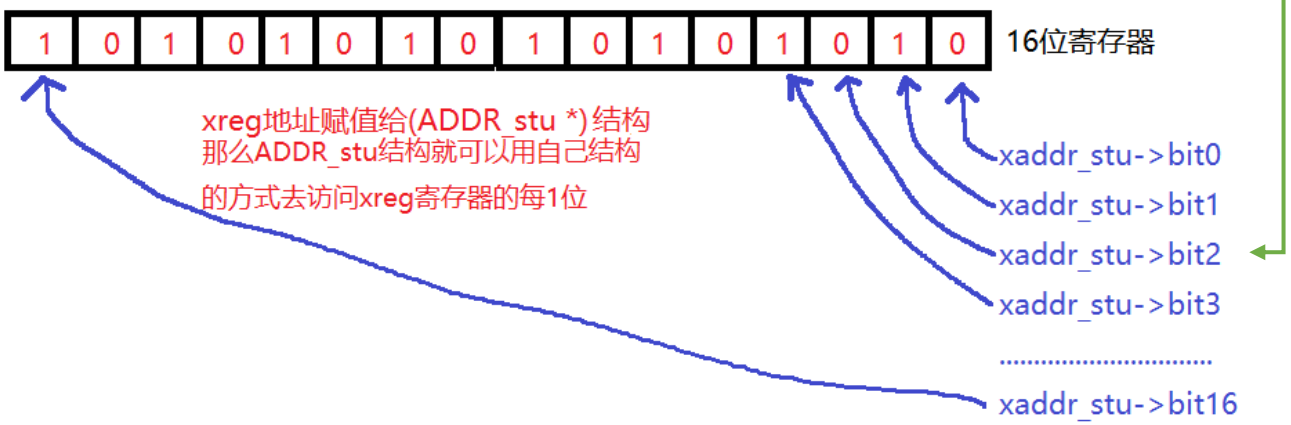
我用这种位域方式就很方便

```
typedef struct
{
        volatile unsigned short bit0 : 1;
        volatile unsigned short bit1 : 1;
        volatile unsigned short bit2 : 1;
        volatile unsigned short bit3 : 1;
        volatile unsigned short bit4 : 1;
        volatile unsigned short bit5 : 1;
        volatile unsigned short bit6 : 1;
        volatile unsigned short bit7 : 1;
        volatile unsigned short bit8 : 1;
        volatile unsigned short bit9 : 1;
        volatile unsigned short bit10 : 1;
        volatile unsigned short bit11 : 1;
        volatile unsigned short bit12 : 1;
        volatile unsigned short bit13 : 1;
        volatile unsigned short bit14 : 1;
        volatile unsigned short bit15 : 1;

}ADDR_stu;
```

一定要用 volatile 来修饰，不然编译器在编译的时候很可能把 unsigned short xx : 1 位运算，优化成常规的 unsigned short 16 位变量

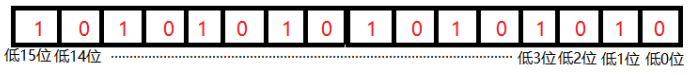
```c
int main(void)
{
        unsigned short xreg = 0xAAAA; //1010 1010 1010 1010 //short 模拟标准16寄存器
        ADDR_stu *xaddr_stu;
        xaddr_stu = (ADDR_stu *)&xreg;

        printf("bit0 = %x\n",xaddr_stu->bit0);
        printf("bit1 = %x\n",xaddr_stu->bit1);
        printf("bit2 = %x\n",xaddr_stu->bit2);
        printf("bit3 = %x\n",xaddr_stu->bit3);
```

将我们模拟的寄存器 xreg 地址强制转换成位域赋值给位域变量



我们打印看看是否将寄存器转换到结构体位域成功

```
bit0 = 0
bit1 = 1
bit2 = 0
bit3 = 1
```

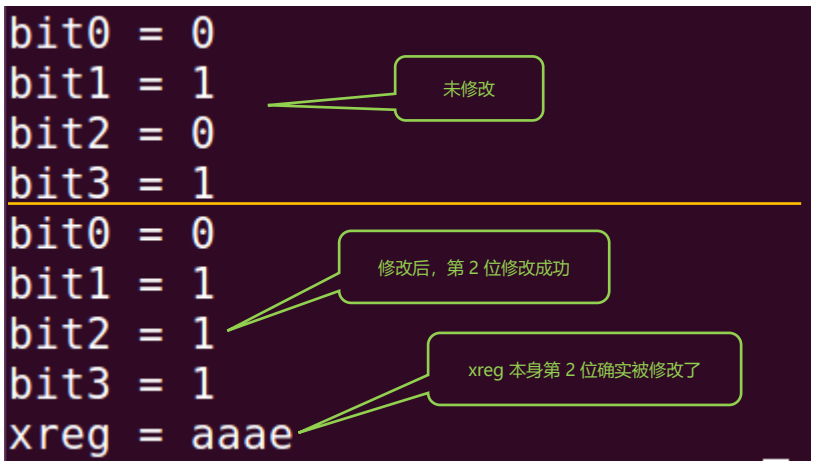


对比查看打印正确

我们写 1 位进去看看是不是对的呢?

```c
int main(void)
{
        unsigned short xreg = 0xAAAA; //1010 1010 1010 1010 //short 模拟标准16寄存器
        ADDR_stu *xaddr_stu;
        xaddr_stu = (ADDR_stu *)&xreg;

        printf("bit0 = %x\n",xaddr_stu->bit0);
        printf("bit1 = %x\n",xaddr_stu->bit1);
        printf("bit2 = %x\n",xaddr_stu->bit2);
        printf("bit3 = %x\n",xaddr_stu->bit3);

        xaddr_stu->bit2 = 1;    //xreg变量第2位写1

        printf("bit0 = %x\n",xaddr_stu->bit0);
        printf("bit1 = %x\n",xaddr_stu->bit1);
        printf("bit2 = %x\n",xaddr_stu->bit2);
        printf("bit3 = %x\n",xaddr_stu->bit3);

        printf("xreg = %x\n ",xreg);//打印变量的值，查看变量是否修改成功
        return 0;
}
```

修改变量里面的某 1 位



0xaaae = 1010 1010 1010 1110　你看 xreg 变量第 2 位被修改

这种方式用在单片机寄存器操作就很方便了，我可以直接放弃移位运算(1<<2)这种麻烦的操作。

下面操作 STM32 单片机 GPIO 管脚的某一位就可以用这种位域方法

```
typedef struct
{
        int CRL;
        int CRH;
        int IDR;
        int ODR;
        int BSRR;
        int BRR;
        int LCKR;

} GPIO_TypeDef;


#define GPIOA_BASE (0x40000000+0x10000+0x0800)  //GPIOA起始地址
#define GPIOA ((GPIO_TypeDef *) GPIOA_BASE) //将起始地址后面的地址转换成可以用结构体名字访问的类型
```

```c
typedef struct
{
	volatile unsigned short bit0 : 1;
	volatile unsigned short bit1 : 1;
	volatile unsigned short bit2 : 1;
	volatile unsigned short bit3 : 1;
	volatile unsigned short bit4 : 1;
	volatile unsigned short bit5 : 1;
	volatile unsigned short bit6 : 1;
	volatile unsigned short bit7 : 1;
	volatile unsigned short bit8 : 1;
	volatile unsigned short bit9 : 1;
	volatile unsigned short bit10 : 1;
	volatile unsigned short bit11 : 1;
	volatile unsigned short bit12 : 1;
	volatile unsigned short bit13 : 1;
	volatile unsigned short bit14 : 1;
	volatile unsigned short bit15 : 1;

}GPIO_bit;

#define PORTA_OUT ((GPIO_bit *)(&(GPIOA->ODR)))

int main(void)
{
	PORTA_OUT->bit0 = 1;

	return 0;
}
```

因为 GPIOA 的 ODR 是 16 位的寄存器，所以我们可以用 unsigned short 来转换

这就是操作 GPIOA bit0 位的方法，因为地址要在 STM32 上才能访问， 所以这里只能浏览代码结构。

```c
#define PORTA_OUT     ((GPIO_Bit_TypeDef *)(&(GPIOA->ODR)))
#define PORTA_IN      ((GPIO_Bit_TypeDef *)(&(GPIOA->IDR)))
#define PORTB_OUT     ((GPIO_Bit_TypeDef *)(&(GPIOB->ODR)))
#define PORTB_IN      ((GPIO_Bit_TypeDef *)(&(GPIOB->IDR)))
#define PORTC_OUT     ((GPIO_Bit_TypeDef *)(&(GPIOC->ODR)))
#define PORTC_IN      ((GPIO_Bit_TypeDef *)(&(GPIOC->IDR)))
#define PORTD_OUT     ((GPIO_Bit_TypeDef *)(&(GPIOD->ODR)))
#define PORTD_IN      ((GPIO_Bit_TypeDef *)(&(GPIOD->IDR)))
#define PORTE_OUT     ((GPIO_Bit_TypeDef *)(&(GPIOE->ODR)))
#define PORTE_IN      ((GPIO_Bit_TypeDef *)(&(GPIOE->IDR)))
#define PORTF_OUT     ((GPIO_Bit_TypeDef *)(&(GPIOF->ODR)))
#define PORTF_IN      ((GPIO_Bit_TypeDef *)(&(GPIOF->IDR)))
#define PORTG_OUT     ((GPIO_Bit_TypeDef *)(&(GPIOG->ODR)))
#define PORTG_IN      ((GPIO_Bit_TypeDef *)(&(GPIOG->IDR)))
```

```c
typedef struct
{
    volatile unsigned short bit0 : 1;
    volatile unsigned short bit1 : 1;
    volatile unsigned short bit2 : 1;
    volatile unsigned short bit3 : 1;
    volatile unsigned short bit4 : 1;
    volatile unsigned short bit5 : 1;
    volatile unsigned short bit6 : 1;
    volatile unsigned short bit7 : 1;
    volatile unsigned short bit8 : 1;
    volatile unsigned short bit9 : 1;
    volatile unsigned short bit10 : 1;
    volatile unsigned short bit11 : 1;
    volatile unsigned short bit12 : 1;
    volatile unsigned short bit13 : 1;
    volatile unsigned short bit14 : 1;
    volatile unsigned short bit15 : 1;

} GPIO_Bit_TypeDef;
```

可以将所有的 GPIO 都定义成这样

#和##连接符使用

#号使用，主要是将宏里面的变量名变成字符显示

```
#define Xconnect(EXP)  #EXP

int main()
{
        printf("%s\n",Xconnect(EXP));

        return 0;
}
```

单#号就是将#后面的英文或者变量值变成字符串形式，这里就是将 EXP 变成字符串

向宏定义传入 EXP

```
root@ubuntu:/home/xzz/test# ./test2
EXP
```

EXP 变成字符被打印出来

```
#include <stdio.h>

#define Xconnect(EXP)  #EXP

int main()
{
        printf("%s\n",Xconnect(50));

        return 0;
}
```

传入数字，也会变成字符被打印出来

```
root@ubuntu:/home/xzz/test# ./test2
50
```

数字变成字符

##连接符

(## 是变量连接符，将两个字符连接成一个变量,<记住是将两个字符连接成一个变量，或者是拼接成值>，而不是连接两个字符串)

##将前后两个参数进行拼接 (比如 100#80 = 10080)

```
#include <stdio.h>

#define XCOMMAND1(NAME) 100##NAME

int main()
{

        printf("%d\n",XCOMMAND1(80));
        return 0;
}
```

将 100 和 NAME 变量传入的值拼接起来

100##80 拼接后 10080

```
root@ubuntu:/home/xzz/test# ./test2
10080
```

在单片机访问寄存器中配合上一节位域的概念，就可以操作寄存器中某一位。

```
#define PORTA_OUT ((GPIO_Bit_TypeDef *)(&(GPIOA->ODR)))
#define PORTA_IN ((GPIO_Bit_TypeDef *)(&(GPIOA->IDR)))
#define PAout(n) (PORTA_OUT->bit##n) //这样用连接符就可以操作某位的高低电平
#define PAin(n) (PORTA_IN->bit##n)
```

如: PAout(1) = 1；就相当于 PORTA_OUT - >bit1 ，表示 GPIOA 1 输出高电平

#define GPIO_SET(X) (X##_PORT)->BSRR = ((uint32_t)1 << X##_PIN) //比如这种？怎么理解？

GPIO_SET(LED_PDM); //这是 GPIO_SET 宏真正的操作方式

这种操作方式需要满足以下几个条件

#define LED_PDM_PORT GPIOB //GPIOB 地址
#define LED_PDM_PIN 1
宏定义了 GPIO 地址和引脚

GPIO_SET(LED_PDM); =

#define GPIO_SET(X) (X##_PORT)->BSRR = ((uint32_t)1 << X##_PIN) 替换如下

代入 LED_PDM_PORT->BSRR = ((uint32_t)1 << LED_PDM _PIN)
就是将 LED_PDM 拼接进去了，不用先定义 LED_PDM 变量，因为是连接符，所以就是传入的字符

```c
#include <stdio.h>

#define LED_PDM_PORT "xLED_PDM_PORT\n"

#define GPIO_SET(X) (X##_PORT)


int main(void)
{
        printf( GPIO_SET(LED_PDM) );
        return 0;
}
```

```
root@ubuntu:/home/xzz/test# ./test
xLED_PDM_PORT
```
运行成功

这种 LED_PDM_PORT 事先就定义好了的，只是为了方便用连接符来调用。

```
#define PWM_START(X)                    \
    X##_TM->BDTR |= TIM_BDTR_MOE; \
    X##_TM->CR1 |= TIM_CR1_CEN
```

PWM_START(CWD_NEN); //就是这种使用方法

我事先在宏里面定义了 CWD_NEN 连接的几种可能性
#define CWD_NEN_PORT GPIOA
#define CWD_NEN_PIN 0
#define CWD_NEN_CC 3
#define CWD_NEN_CH 3
#define CWD_NEN_AF 4
#define CWD_NEN_TM TIM2

如果 PWM_START(CWD_NEN); 就相当于如下：

#define PWM_START(CWD_NEN) //启动 TIM2
    CWD_NEN _TM->BDTR |= TIM_BDTR_MOE;
    CWD_NEN _TM->CR1 |= TIM_CR1_CEN

    就是替换成 TIM2
#define PWM_START(CWD_NEN)
    TIM2->BDTR |= TIM_BDTR_MOE;
    TIM2->CR1 |= TIM_CR1_CEN

如果我用 TIM14 呢?
#define WWD_NEN_TM TIM14

那就是: PWM_START(WWD_NEN);

#define PWM_START(WWD_NEN) //启动 TIM14
    TIM14->BDTR |= TIM_BDTR_MOE;
    TIM14->CR1 |= TIM_CR1_CEN

这就是连接符方便调用寄存器的方法，用人容易看懂的方式去使用寄存器，只不过代价就是要预先写很多这种宏。

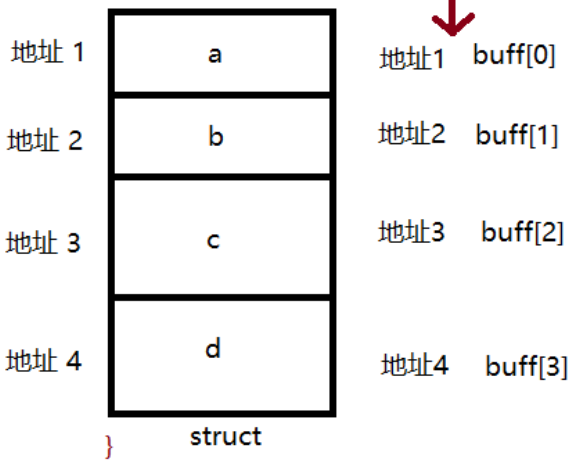
联合体配合结构体和一维数组的高级应用

```
typedef union
{
        struct
        {
                unsigned int a;
                unsigned int b;
                unsigned int c;
                unsigned int d;
        };

        unsigned int buff[4];

}statusInfo;
```

如果我要读写结构体里面某个变量，可以用数组的方式去访问

联合体内存布局如下：

typedef union{



联合体，访问另外数组变量，访问的也是结构体同样的地址

地址1 buff[0]

地址2 buff[1]

地址3 buff[2]

地址4 buff[3]

这就是为什么联合体也叫做公用体，就是可以用两种变量名访问同一片内存，但是这有什么好处呢？

```
int main()
{
        statusInfo value;

        value.buff[0] = 10;
        value.buff[1] = 20;
        value.buff[2] = 30;
        value.buff[3] = 40;

        printf("a = %d\n", value.a);
        printf("b = %d\n", value.b);
        printf("c = %d\n", value.c);
        printf("d = %d\n", value.d);

        return 0;
}
```

比如我向数组里面写数据

```
unsigned int buff[4];
```

其实也就是向同一片内存结构体写数据

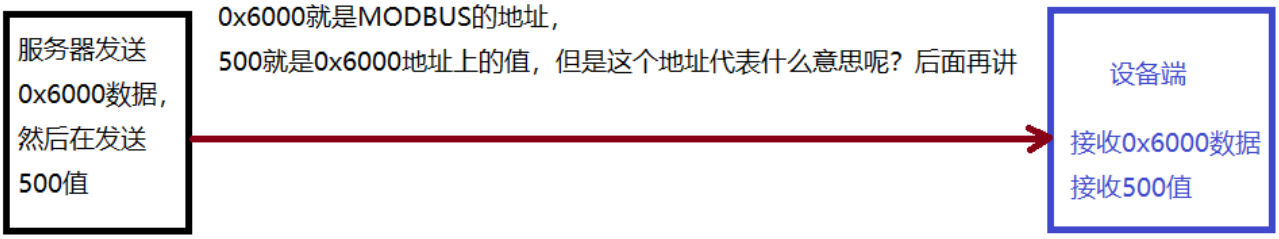
```
struct
{
        unsigned int a;
        unsigned int b;
        unsigned int c;
        unsigned int d;
};
```

```
a = 10
b = 20
c = 30
d = 40
```

读出来看看，其实就是我 value.buff 写入的数据

这样做有什么好处？用两个不同的变量来访问同一个内存空间？

比如在单片机 MODBUS TCP 协议的应用中



在这种情况下你不知道地址0x6000是代表的什么意思
所以我要给地址再取个名字，比如0x6000表示 ' 长度 '

```
typedef union
{
        struct
        {
                unsigned int Size;
                unsigned int width;
                unsigned int radius;
                unsigned int diameter;
        /******要定义很多int 一直定义到6000(24576)个int才结束*********/
        /*******.........................................*********/
                unsigned int Length;//最后这个变量就是0x6000(24576)位置 长度
        };

        unsigned int buff[25476];//数组也一定要和int数量一致，这样数组最后一个元素就是长度

}statusInfo;
```

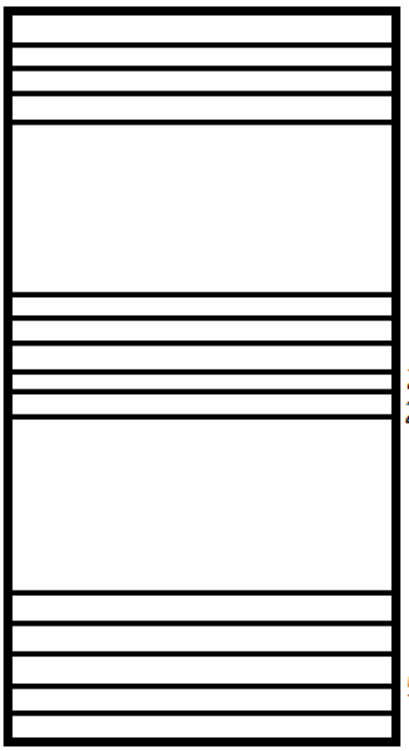
下面来讲解如何使用这个数据结构

```
int main()
{
        statusInfo value;
        Recv(地址数据，值) //比如接收到网络数据
        value.buff[地址数据] = 值;//将接收到的地址放入数组下标，值放入数组
        value.buff[24576] = 值; //比如我就是接收的24576地址，那么就把24576放入数组
        ret = value.Length;//我不知道24576到底代表的什么，我就用对应的结构体来获取
        /*我们知道了24576就是代表的长度length*/
```

这就是设备端为了人类看得懂，所以用结构体变量来获取 MODBUS 发过来的值。
如果服务器一次性发送 1000 个 MODBUS 的数据，你不能定义 1000 个变量去接收，你最好用数组去接收，然后用结构体来获取。

联合体在 MODBUS 协议应用的时候，寄存器排列方式注意事项

有时候会遇到一些需求，MODBUS地址使用不连续



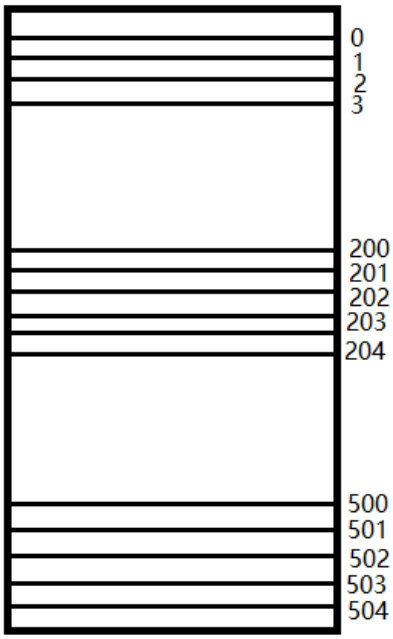
第一组MODBUS寄存器只使用0，1，2，3地址

第二组MODBUS寄存器不是跟着第4地址使用的，而是直接跳转到200地址开始使用

第三组MODBUS寄存器也不是跟着第205个地址使用的，而是直接跳转到500地址开始使用

MODBUS寄存器地址

这样的话MODBUS寄存器结构体和数组该怎么写?



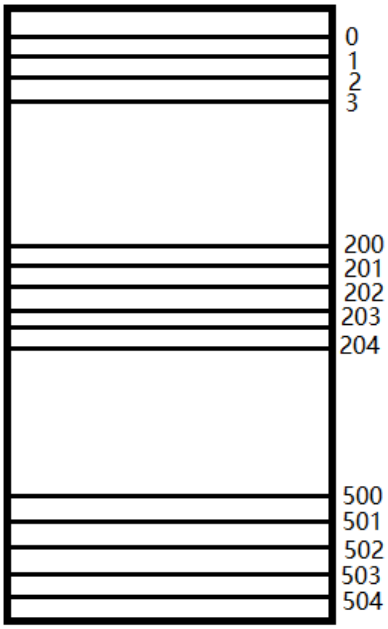
数组好办

```
int array[505];
```

直接给数组定义505个元素就是了，只是有点浪费内存空间，因为只有0~3地址的元素，200~204地址的元素和500~504地址的元素是有使用的，数组其余的元素都是空起

那么结构体定义MODBUS寄存器又怎么办呢?

MODBUS寄存器地址



MODBUS寄存器地址

```
typedef struct
{
        int addr0;
        int addr1;
        int addr2;
        int addr3;
        int reserved1[197];
        int addr200;
        int addr201;
        int addr202;
        int addr203;
        int addr204;
        int reserved2[296];
        int addr500;
        int addr501;
        int addr502;
        int addr503;
        int addr504;
}MODBUS_reg
```

modbus地址跳过的部分用默认数组来填充，这里200地址和3地址相差200-3 = 197，所以默认数组填充197个元素，后面定义的int就从200开始排列。

但是请注意，数组好像是从0开始算的

所以 int addr200；地址有可能是201，实测为准

你看，在项目中我要访问 200 的地址，数组最后一个下标 200 地址还不对，因为数组是从 0 开始计算的，所以数组最后 200 下标的地址可能是 199。所以数组后面定义的变量才是 200 地址

```
typedef struct
{
  uint16_t reserved1[200];              //必须从200后面跟第一个硬件版本是200，因为数组是从0开始算的
  uint16_t Hardware_version;            //200 硬件版本 r
  uint16_t Software_version;            //201 软件版本 r
  uint16_t WorkStatus;                  //202 工作状态
  uint16_t BatVoltage;                  //203 电池电压
  uint16_t DI_channel_1_8_data;         //204 DI通道1~8数据
  uint16_t DI_channel_9_16_data;        //205 DI通道9~16数据
```

```
typedef union
{
  MODBUS_reg RegName;
  uint16_t array[326];

}MODBUS_unreg;
```

最后使用枚举的方式操作 MODBUS 寄存器

函数宏(主要是为了降低压栈和弹栈的开销，方便移植。不然直接建立函数来实现更好)

函数宏，即包含多条语句的宏定义，其通常为某一被频繁调用的功能的语句封装，且不想通过函数方式封装来降低额外的弹栈压栈开销。

```
#define INT_SWAP(a,b) \
        int tmp = a;     \
        a = b;           \
        b = tmp


int main(void)
{
        int a = 10 , b = 20;
        INT_SWAP(a,b);
        printf("a = %d  b = %d\n",a,b);

        return 0;
}
```

```
root@ubuntu:/home/xzz/test# ./test
a = 20  b = 10
```

两个变量交互值。这种宏方式当使用 if、while 等语句且不使用花括号，直接调用宏 INT_SWAP 时程序运行到第 1 个 分号 int tmp = a; 就结束了。不会运行 a=b; 和 b = tmp

```
#define INT_SWAP(a,b) \
{       |       |       \
        int tmp = a;     \
        a = b;           \
        b = tmp;         \
}       |       |       \

int main(void)
{
        int a = 10 , b = 20;
        INT_SWAP(a,b);
        printf("a = %d  b = %d\n",a,b);

        return 0;
}
```

```
root@ubuntu:/home/xzz/test# ./test
a = 20  b = 10
```

加入花括号之后也能正常运行

```
#define INT_SWAP(a,b) \
do{     |       |       \
        int tmp = a;     \
        a = b;           \
        b = tmp;         \
}while(0)       |       |

int main(void)
{
        int a = 10 , b = 20;
        INT_SWAP(a,b);
        printf("a = %d  b = %d\n",a,b);

        return 0;
}
```

```
root@ubuntu:/home/xzz/test# ./test
a = 20  b = 10
```

do{...}while(0) 表示只执行一遍 {} 内的语句

用 do.... while 有什么好处呢?

由于 do{...}while(0) 实际为 while 循环，因此可以使用关键字 break 提前结束循环。利用该特性，可以为函数宏添加参数检测。例如：

```
#define INT_SWAP(a,b) \
do{                   \
        if ( a < 0 || b < 0 ) \
        break;        \
                      \
        int tmp = a;  \
        a = b;        \
        b = tmp;      \
}while(0)

int main(void)
{
        int a = -1 , b = -2;
        INT_SWAP(a,b);
        printf("a = %d  b = %d\n",a,b);

        return 0;
}
```

```
root@ubuntu:/home/xzz/test# ./test
a = -1  b = -2
```

数据没有发生交换，因为发现 a 和 b 都 < 0 所以 在 break 提前退出了

```
do{                   \
        if ( a < 0 || b < 0 ) \
        break;        \
```

提前退出

```
#define INT_SWAP(a,b) \
do{                   \
        if ( a < 0 || b < 0 ) \
        break;        \
                      \
        int tmp = a;  \
        a = b;        \
        b = tmp;      \
}while(0)

int main(void)
{
        int a = 50 , b = 100;
        INT_SWAP(a,b);
        printf("a = %d  b = %d\n",a,b);

        return 0;
}
```

```
root@ubuntu:/home/xzz/test# ./test
a = 100  b = 50
```

a 和 b 不小于 0，数据交换

如果我想让函数宏运行完之后有返回值怎么做呢?

有些项目中用函数宏做判断，然后返回判断结果，以供其它程序识别

（{}）大括号外加小括号 ， 为 GNU C 扩展的语法，非 C 语言的原生语法。

与 do{...}while(0) 不同的是，（{}）不能提前退出函数宏

所以({})可以当做函数来用，还可以返回值

```c
#define INT_SWAP(a,b) \
({ \
        int ret = 0; \
        if ( a < 0 || b < 0 ) \
        { \
                ret = -1; \
        } \
        else \
        { \
                int tmp = a; \
                a = b; \
                b = tmp; \
        } \
        ret; \
})

int main(void)
{
        int a = -1 , b = -1;
        int ret = 0;
        ret = INT_SWAP(a,b);
        printf("ret = %d\n",ret);
        printf("a = %d  b = %d\n",a,b);

        return 0;
}
```

这种有({})括号的宏，里面的代码可以当做函数来写，但是记住不能用 return 退出哦

如果传入的值小于 0，ret 为-1

如果传入的值大于 0，ret 为默认值，就是 int ret = 0 这句

函数结尾执行的变量就是返回值

```
root@ubuntu:/home/xzz/test# ./test
ret = -1
a = -1  b = -2
```

我将 b 改成了-2，函数宏返回-1，证明 a 和 b < 0 没有交换

```c
#define INT_SWAP(a,b) \
({ \
        int ret = 0; \
        if ( a < 0 || b < 0 ) \
        { \
                ret = -1; \
        } \
        else \
        { \
                int tmp = a; \
                a = b; \
                b = tmp; \
        } \
        ret; \
})

int main(void)
{
        int a = 1000 , b = 2000;
        int ret = 0;
        ret = INT_SWAP(a,b);
        printf("ret = %d\n",ret);
        printf("a = %d  b = %d\n",a,b);

        return 0;
}
```

```
root@ubuntu:/home/xzz/test# ./test
ret = 0
a = 2000  b = 1000
```

a 和 b 大于 0，返回 0，变量进行交换

钩子函数使用方式

钩子函数就是函数指针的一种，也是为了完成简单的分层，让一个功能可以方便多个人开发的一种设计模式。

```c
#include <stdio.h>

int max(int a, int b){
    if(a > b)
        return a;
    return b;
}

int min(int a, int b){
    if(a > b)
        return b;
    return a;
}

int main( int argc, char *argv[] )
{

    int (*pFun)(int x, int y); //函数指针
    int a = 1;
    int b = 2;
    int ret;//定义一个变量来接收调用函数后的返回值

    pFun = max;
    ret = pFun(a, b);//相当于直接调用 max(a,b);
    printf("%d\n", ret);
    return 0;
}
```

一般这样使用函数指针没有什么意义

一般需要两个 C 文件，比如 A.c 和 B.c，一个工程师维护 A.c 文件，一个工程师维护 B.c 文件。

比如 A.c 实现

```c
#include <stdio.h>

int (*global_pFun)(int a, int b);//由于后面要在注册函数中进行挂钩操作，所以把这个函数指针定义成全局变量

int RegFun(int (*pFun)(int a, int b)){ //一定是先注册，这个函数交个头文件
    global_pFun = pFun;
    return 0;
}

int main( int argc, char *argv[] )
{

    int a = 1;
    int b = 2;
    int ret;
    ret = global_pFun(a, b);//直接使用这个钩子函数，函数内部怎么实现的，交由 B 工程师
    printf("%d\n", ret);
    return 0;
}
```

一定要等 B 工程师注册之后，再运行钩子函数，不然会出现空函数 BUG

现在 B 工程师实现

```
#include "A.h" //包含 A 工程师提供的 h 文件接口

//B 工程师自己实现的子函数
int max(int a, int b){
    if(a > b)
        return a;
    return b;
}

int main( int argc, char *argv[] )
{

    int a = 1;
    int b = 2;
    int ret;
    RegFun(max); //注册 B 工程师实现的函数
    //调用 A 工程师实现的 main 函数.......

    return 0;
}
```

这就是钩子函数的意义，一定是定义全局函数指针，然后让 B 工程师实现具体内容，A 工程师再去调用 B 工程师实现的功能。这种方式的好处就是 A 工程师可以先把钩子函数实现起走，然后 A 工程师可以先调用钩子函数的空函数，等 B 工程师有时间了，把 B 工程师实现的功能注册进来。

C 语言实现 MVC 模式实现多人开发

首先，我们需要定义三个主要组件：模型(model)、视图(view)和控制器(controller)。每个组件都有自己的职责和相应的操作。

1. 模型(model): 模型包含了程序所处理的数据结构和算法。它负责存储和管理数据，并提供对这些数据进行修改或查询的接口。下面是一个示例的模型部分的代码：

```
typedef struct {
    int data; // 数据成员
} Model;

void model_setData(Model* m, int value) {
    m->data = value;
}

int model_getData(const Model* m) {
    return m->data;
}
```

2. 视图(view): 视图负责将模型的数据展示给用户。它会从模型获取数据，然后根据需求显示到界面上。

下面是一个示例的视图部分的代码

```
#include <stdio.h>
void view_displayData(const Model* m) {
    printf("The current data is %d\n", model_getData(m));
}
```

3. 控制器(controller): 控制器负责处理用户输入和更新模型。当用户与界面交互时，控制器会调用合适的模型函数来更新数据。下面是一个示例的控制器部分的代码

```
void controller_updateData() {
    Model m;

    // 初始化模型数据为 0
    model_setData(&m, 0);

    // 模拟用户输入
    char input[5];
    printf("Enter a new value for the data (max 4 digits): ");
    scanf("%s", input);

    // 转换字符串为整数并设置到模型中
    if (sscanf(input, "%d", &m.data) == 1 && m.data >= 0 && m.data <= 9999) {
        printf("New data set to %d\n", m.data);

        // 更新完数据后，调用视图函数显示最新的数据
        view_displayData(&m);
    } else {
        printf("Invalid input!\n");
    }
}
```

以上就是 C 语言实现的最基本 MVC 模式方法。

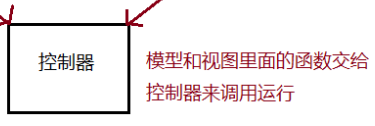
能多人开发的 MVC 代码编写方式

定义 model.h - 模型接口

```
#ifndef MODEL_H
#define MODEL_H

typedef struct Model Model;

// 初始化模型
Model* createModel();

// 更新模型数据
void updateModelData(Model* model, const char* newName, int newAge);

// 获取模型数据
void getModelData(const Model* model, char* name, int* age);

// 销毁模型
void destroyModel(Model* model);

#endif // MODEL_H
```

定义view.h - 视图接口

```
#ifndef VIEW_H
#define VIEW_H

typedef struct View View;

// 初始化视图
View* createView();

// 显示模型数据
void displayModelData(View* view, const char* name, int age);

// 销毁视图
void destroyView(View* view);

#endif // VIEW_H
```



代码实现：model.c - 模型实现

```c
#include "model.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct {
    char name[50];
    int age;
} ModelImpl;
//创建模型
Model* createModel() {
    ModelImpl* model = (ModelImpl*)malloc(sizeof(ModelImpl));
    if (model) {
        model->age = 0;
        model->name[0] = '\0';
    }
    return (Model*)model;
}
//给模型写入数据
void updateModelData(Model* model, const char* newName, int newAge) {
    strncpy(model->name, newName, sizeof(((ModelImpl*)model)->name) - 1);
    model->age = newAge;
}
 //读取模型里面的数据
void getModelData(const Model* model, char* name, int* age) {
    strncpy(name, model->name, sizeof(model->name));
    *age = model->age;
}
void destroyModel(Model* model) { //销毁模型
    free(model);
```

代码实现：view.c - 视图实现

```c
#include "view.h"
#include <stdio.h>

typedef struct {
    // 视图实现所需的数据结构
} ViewImpl;

View* createView() {
    // 初始化视图
    ViewImpl* view = malloc(sizeof(ViewImpl));
    // ...
    return (View*)view;
}

void displayModelData(View* view, const char* name, int age) {
    // 在视图中显示模型数据
    printf("Name: %s\n", name);
    printf("Age: %d\n", age);
}

void destroyView(View* view) {
    // 销毁视图
    free(view);
}
```

这里就要传入模型的数据，怎么得到模型数据传入呢？这就交给控制器。

controller.c - 控制器实现

```c
#include "model.h"  //加入模型
#include "view.h"   //加入视图
#include <stdio.h>

void handleUserInput(Model* model, View* view) {
    char newName[50];
    int newAge;
    strcpy(newName, "XXZZTT");  //设置数据名字
    newAge = 20;  //设置数据值
    updateModelData(model, newName, newAge);
    char modelName[50];
    int modelAge;
    getModelData(model, modelName, &modelAge);  //将模型里面的值获取出来，控制器做的事。
    displayModelData(view, modelName, modelAge); //将获取的数据传入给视图显示。控制器做的事
}
        //工程师B只需要实现这个函数就是了。
int main() {
    Model* model = createModel();  //创建模型
    View* view = createView(); //创建视图

    handleUserInput(model, view);  //使用控制器

    destroyModel(model);
    destroyView(
}
```

typedef struct Model Model;

传入之前头文件定义的model，然后给model赋值。这个可以交给工程师A，在工程师A的函数里面实现。