

知识点 1 【结构体的概述】

类型分为：

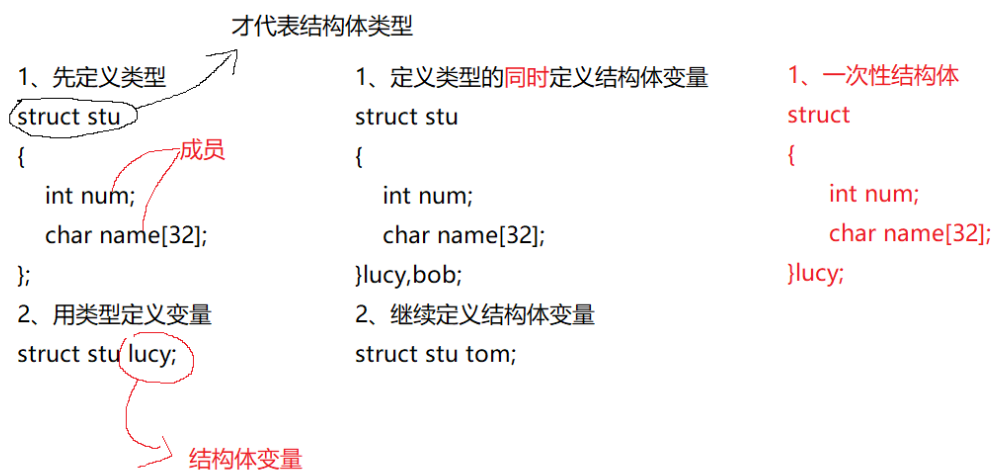
1、基本类型 char short int long float double

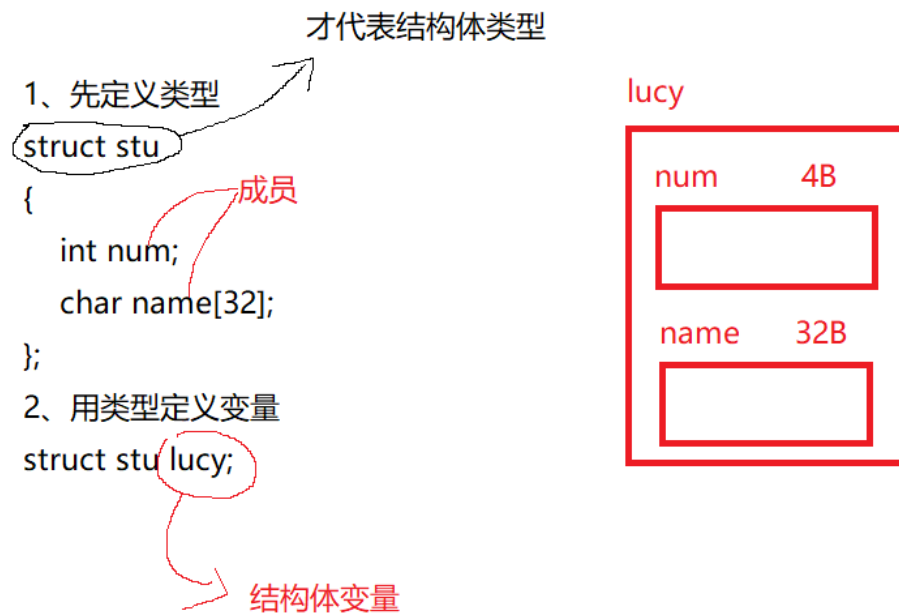
2、构造类型：有基本类型封装打包而来，char name[32]; 结构体 struct 共有体

union 枚举 enum

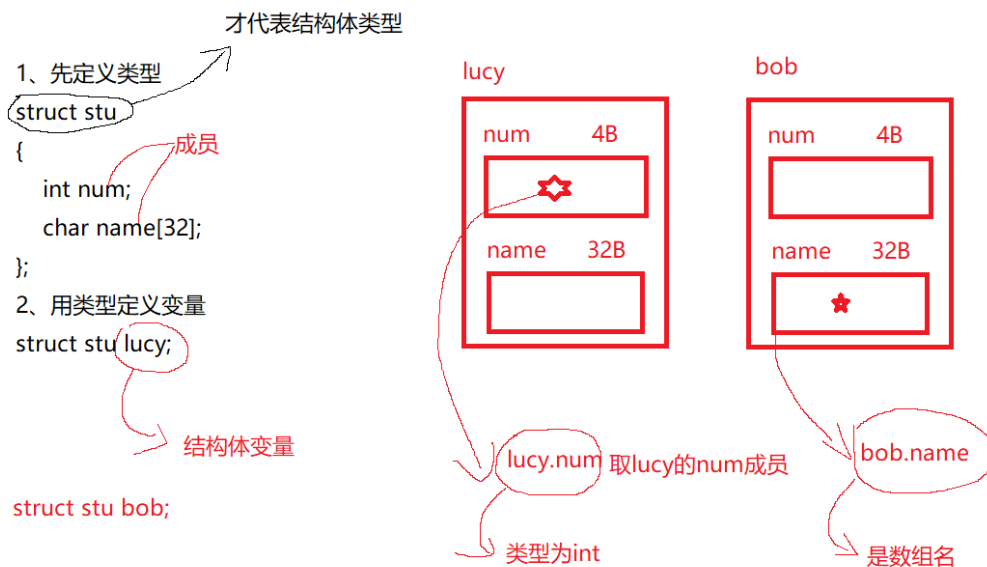
结构体：有关键字 struct 修饰，是一种或多种基本类型或构造类型的数据的集合。

1、结构体类型的定义





结构体的成员拥有独立的空间。



2、定义结构体变量

操作结构体成员的时候 必须遵循 成员自身的类型。 (重要)

```

3 // struct stu 才是结构体的完整类型
4 struct stu
5 {
6     int num; // 成员 定义类型的时候 不要给成员赋值
7     char name[32];
8     float score;
9 };
10
11 void test01()
12 {
13     struct stu lucy; // lucy就是结构体变量
14     // 访问成员的时候必须尊许 成员自身的类型
15     printf("%d %s %f\n", lucy.num, lucy.name, lucy.score);
16 }

```

3、结构体变量的初始化

结构体变量初始化时 必须尊许成员自身类型以及成员顺序。

```

3 // struct stu 才是结构体的完整类型
4 struct stu
5 {
6     int num; // 成员 定义类型的时候 不要给成员赋值
7     char name[32];
8     float score;
9 };
10
11 void test01()
12 {
13     // 结构体变量初始化时 必须遵循成员自身类型以及成员顺序
14     struct stu lucy = {100, "lucy", 88.8f};
15
16     printf("%d %s %f\n", lucy.num, lucy.name, lucy.score);
17 }
18 int main(int argc, char const *argv[])
19 {

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```

● edu@edu:~/work/c/day08$ gcc 00_code.c
● edu@edu:~/work/c/day08$ ./a.out
0 0.000000
● edu@edu:~/work/c/day08$ gcc 00_code.c
● edu@edu:~/work/c/day08$ ./a.out
100 lucy 88.800003
○ edu@edu:~/work/c/day08$

```

```

4  struct stu
5  {
6      int num; // 成员 定义类型的时候 不要给成员赋值
7      char name[32];
8      float score;
9  };
10 void test01()
11 {
12     // 结构体变量初始化时 必须遵循成员自身类型以及成员顺序
13     struct stu lacy = {100, "lacy", 88.8f};
14     struct stu bob;
15     memset(&bob, 0, sizeof(bob)); // 将结构体变量清0
16
17     printf("%d %s (char [10]) %d %s %f\n", bob.num, bob.name, bob.score);
18     printf("%d %s %f\n", bob.num, bob.name, bob.score);
19 }

```

4、结构体成员的操作

操作结构体成员的时候 必须遵循 成员自身的类型。 (**重要**)

```

#include <stdio.h>

#include <string.h>

// struct stu 才是结构体的完整类型

struct stu

{

    int num; // 成员 定义类型的时候 不要给成员赋值

    char name[32];

    float score;

};

void test01()

{

    // 结构体变量初始化时 必须遵循成员自身类型以及成员顺序

    struct stu lacy = {100, "lacy", 88.8f};

```

```

struct stu bob;

memset(&bob, 0, sizeof(bob)); // 将结构体变量清 0


// 将 lucy 内容 赋值 给 bob (方法 1: 逐个成员操作)

// bob.num = lucy.num;

// strcpy(bob.name, lucy.name);

// bob.score = lucy.score;


// 将 lucy 内容 赋值 给 bob (方法 2: 相同类型的结构体变量可以=赋值,浅拷贝)

// bob = lucy;


// 将 lucy 内容 赋值 给 bob (方法 3: memcpy)

memcpy(&bob, &lucy, sizeof(struct stu));


printf("%d %s %f\n", lucy.num, lucy.name, lucy.score);

printf("%d %s %f\n", bob.num, bob.name, bob.score);

}

```

```

● edu@edu:~/work/c/day08$ gcc 00_code.c
● edu@edu:~/work/c/day08$ ./a.out
100 lucy 88.800003
100 lucy 88.800003
○ edu@edu:~/work/c/day08$ █

```

5、键盘给结构体成员赋值

操作结构体成员的时候 必须遵循 成员自身的类型。 (重要)

```
32 void test02()
33 {
34     struct stu lucy;
35     memset(&lucy, 0, sizeof(struct stu));
36
37     printf("请输入学号 姓名 分数:");
38     scanf("%d %s %f", &lucy.num, lucy.name, &lucy.score);
39
40     printf("%d %s %f\n", lucy.num, lucy.name, lucy.score);
41 }
42 int main(int argc, char const *argv[])
43 {
44     test02();
45 }
```

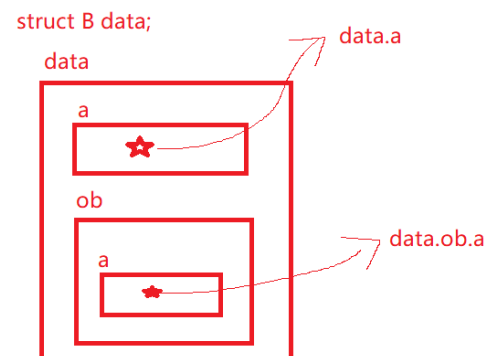
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
edu@edu:~/work/c/day08$ gcc 00_code.c
edu@edu:~/work/c/day08$ ./a.out
请输入学号 姓名 分数:100 lucy 99.9
100 lucy 99.900002
edu@edu:~/work/c/day08$
```

6、结构体嵌套结构体

访问成员的时候 一定要访问到最底层。

```
struct A
{
    int a;
};
struct B
{
    int a;
    struct A ob; //结构体嵌套了结构体
};
```



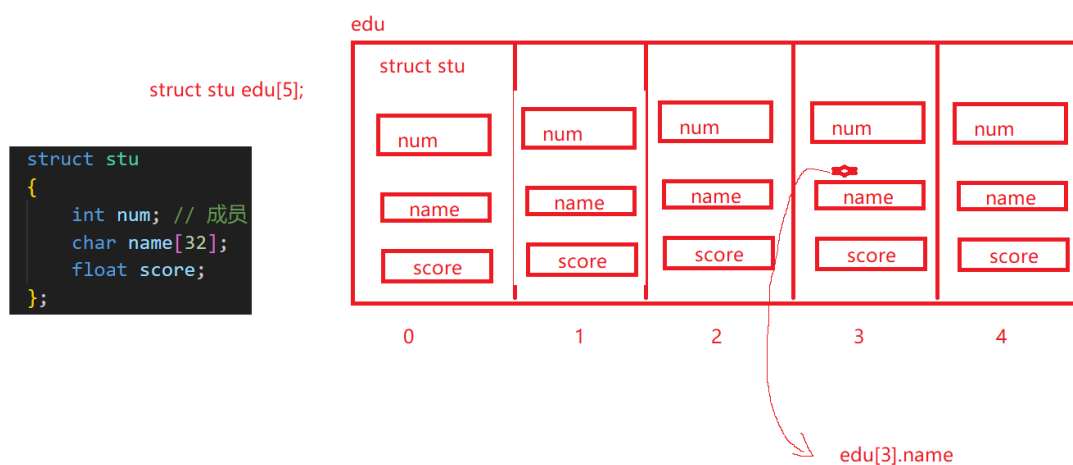
```
43 struct A
44 {
45     int a;
46 };
47 struct B
48 {
49     int a;
50     struct A ob; // 结构体嵌套了结构体
51 };
52 void test03()
53 {
54     struct B data = {100, {200}};
55     printf("%d %d\n", data.a, data.ob.a);
56 }
57 int main(int argc, char const *argv[])
58 {
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
edu@edu:~/work/c/day08$ gcc 00_code.c
edu@edu:~/work/c/day08$ ./a.out
100 200
edu@edu:~/work/c/day08$
```

知识点 2【结构体数组】

结构体数组：本质是数组 每个元素是结构体。



```
struct stu

{

    int num; // 成员 定义类型的时候 不要给成员赋值

    char name[32];

    float score;

};

void input_struct_stu_array(struct stu *arr, int n)

{

    printf("请输入%d 个学生信息\n", n);

    int i = 0;

    for (i = 0; i < n; i++)

    {

        scanf("%d %s %f", &arr[i].num, arr[i].name, &arr[i].score);

    }

    return;

}

void print_struct_stu_array(struct stu *arr, int n)

{

    int i = 0;

    for (i = 0; i < n; i++)

    {

        printf("%d %s %f\n", arr[i].num, arr[i].name, arr[i].score);

    }

}
```



```
}

return;

}

void test05()

{

    struct stu edu[5];

    int n = sizeof(edu) / sizeof(edu[0]);

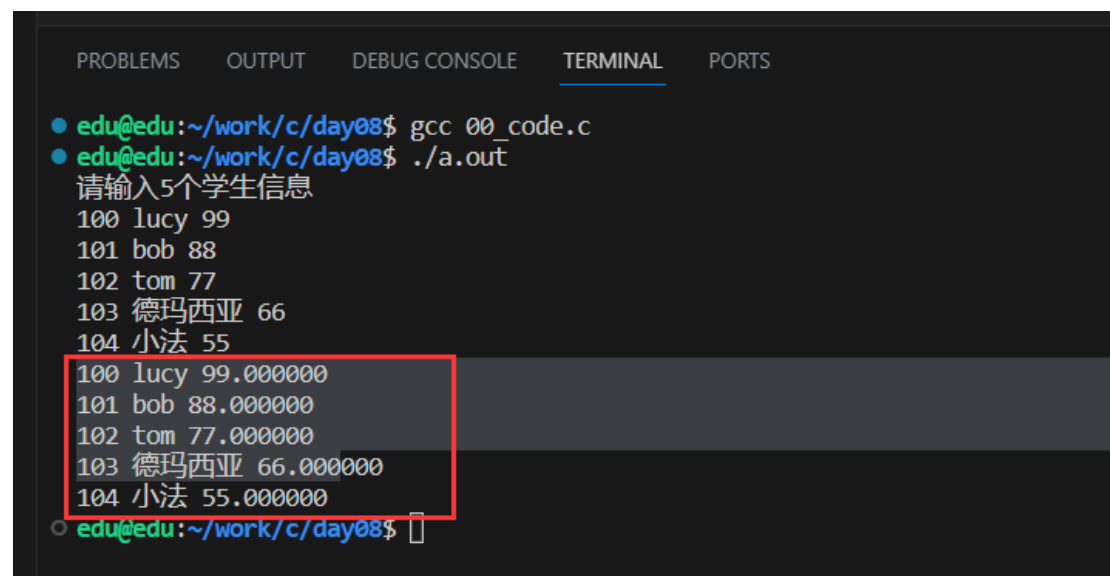
    // 给这个 5 个学员获取数据

    input_struct_stu_array(edu, n);

    // 遍历结构体数组

    print_struct_stu_array(edu, n);

}
```



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

● edu@edu:~/work/c/day08$ gcc 00_code.c
● edu@edu:~/work/c/day08$ ./a.out
请输入5个学生信息
100 lucy 99
101 bob 88
102 tom 77
103 德玛西亚 66
104 小法 55
100 lucy 99.000000
101 bob 88.000000
102 tom 77.000000
103 德玛西亚 66.000000
104 小法 55.000000
○ edu@edu:~/work/c/day08$
```

知识点 3 【指针成员】

如果结构体中有指针成员 一定要记得将**指针成员**指向**合法的空间**（栈区、全局区、文字常量区、堆区）

```
2 typedef struct stu
3 {
4     int num;
5     char *name; // 指针成员 保存的是外部字符串的首元素地址
6     float score;
7 } STU; // STU就是struct stu类型
8 void test01()
9 {
10     STU lucy;
11     printf("%d %s %f\n", lucy.num, lucy.name, lucy.score);
12 }
```

lucy 栈区

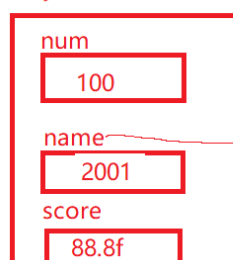


如果lucy没有初始化 name为指针成员指向一个未知区域，%s遍历 可能段错误 可能访问未知的内容

1、指针成员在**栈区** 指向 **栈区**

```
1 #include <stdio.h>
2 typedef struct stu
3 {
4     int num;
5     char *name; // 指针成员 保存的是外部字符串的首元素地址
6     float score;
7 } STU; // STU就是struct stu类型
8 void test01()
9 {
10     //buf在栈区
11     char buf[]="hello world";
12     STU lucy={100,buf,88.8f};
13     printf("%d %s %f\n", lucy.num, lucy.name, lucy.score);
14 }
```

lucy 栈区



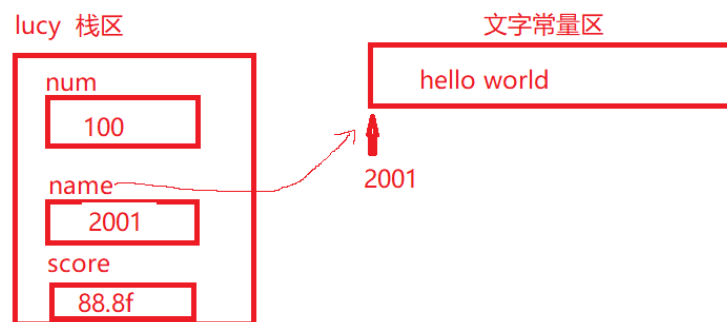
buf 栈区



2001

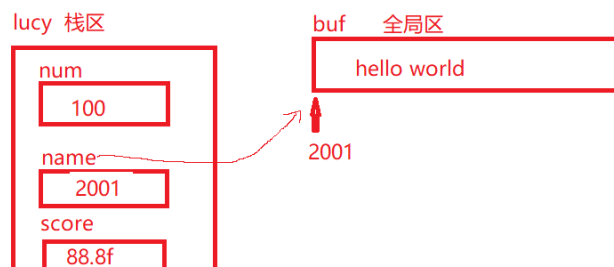
2、指针成员在栈区 指向 文字常量。

```
1  #include <stdio.h>
2  typedef struct stu
3  {
4      int num;
5      char *name; // 指针成员 保存的是外部字符串的首元素地址
6      float score;
7  } STU; // STU就是struct stu类型
8  void test01()
9  {
10     STU lcy={100,"hello world",88.8f};
11     printf("%d %s %f\n", lcy.num, lcy.name, lcy.score);
12 }
```



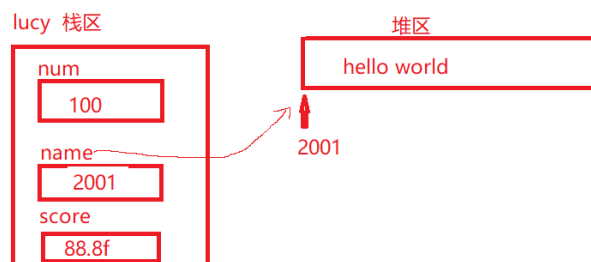
3、指针成员在栈区 指向 全局区。

```
1  #include <stdio.h>
2  typedef struct stu
3  {
4      int num;
5      char *name; // 指针成员 保存的是外部字符串的首元素地址
6      float score;
7  } STU; // STU就是struct stu类型
8  char buf[]="hello world";
9  void test01()
10 {
11     STU lcy={100,buf,88.8f};
12     printf("%d %s %f\n", lcy.num, lcy.name, lcy.score);
13 }
```



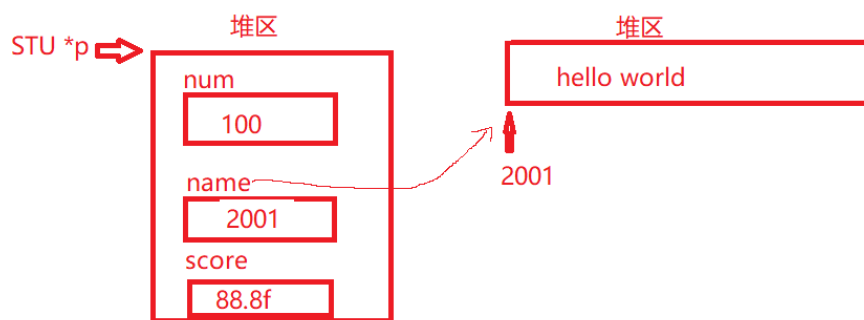
4、指针成员在栈区 指向 堆区。

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 typedef struct stu
5 {
6     int num;
7     char *name; // 指针成员 保存的是外部字符串的首元素地址
8     float score;
9 } STU; // STU就是struct stu类型
10 void test01()
11 {
12     STU lucy;
13     lucy.num = 100;
14     lucy.score = 88.8f;
15     lucy.name = (char *)malloc(32);
16     strcpy(lucy.name, "hello world");
17
18     printf("%d %s %f\n", lucy.num, lucy.name, lucy.score);
19     free(lucy.name);
20 }
```



5、指针成员在堆区 指向 堆区。

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 typedef struct stu
5 {
6     int num;
7     char *name; // 指针成员 保存的是外部字符串的首元素地址
8     float score;
9 } STU; // STU就是struct stu类型
10 void test01()
11 {
12     //结构体在堆区
13     STU *p = (STU *)malloc(sizeof(STU));
14     (*p).num = 100;
15     (*p).name = (char *)malloc(32); //指向在堆区
16     strcpy((*p).name, "hello world");
17     (*p).score = 88.8f;
18
19     printf("%d %s %f\n", (*p).num, (*p).name, (*p).score);
20     free((*p).name);
21     free(p);
22 }
23 int main(int argc, char const *argv[])
24 {
25     test01();
26     return 0;
27 }
28
```



知识点 4 【结构体的浅拷贝和深拷贝】

如果结构体中**没有**指针成员 **浅拷贝** 不会有问题。

如果结构体中**有**指针成员 **浅拷贝** 会有问题。

1、**浅拷贝**：结构体变量的**空间内容** 直接赋值给 另一个**结构体** 变量的**空间**。

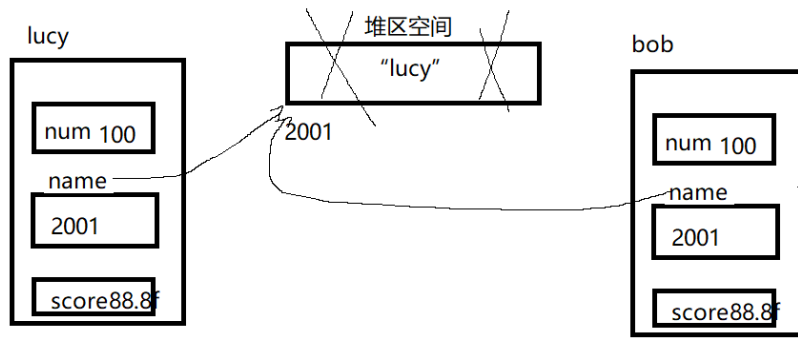
比如：相同类型的结构体变量可以使用=直接赋值（浅拷贝）

```
STU lucy={100,"lucy",88.8f};
```

```
STU bob;
```

```
bob=lucy;//浅拷贝
```

如果结构体中**有**指针成员 且指针成员 **指向了堆区** 这样**浅拷贝**有可能造成内存多次释放问题（重要）



```
if (lucy.name != NULL)
{
    free(lucy.name);
    lucy.name = NULL;
}
```

```
if (bob.name != NULL)
{
    free(bob.name);
    bob.name = NULL;
}
```

```
void test02()
{
    STU lucy;

    lucy.num = 100;

    lucy.name = (char *)malloc(32);

    strcpy(lucy.name, "lucy");

    lucy.score = 88.8f;

    STU bob;

    //浅拷贝

    bob = lucy;

    printf("%d %s %f\n", bob.num, bob.name, bob.score);

    if (lucy.name != NULL)
```

```

{
    free(lucy.name);

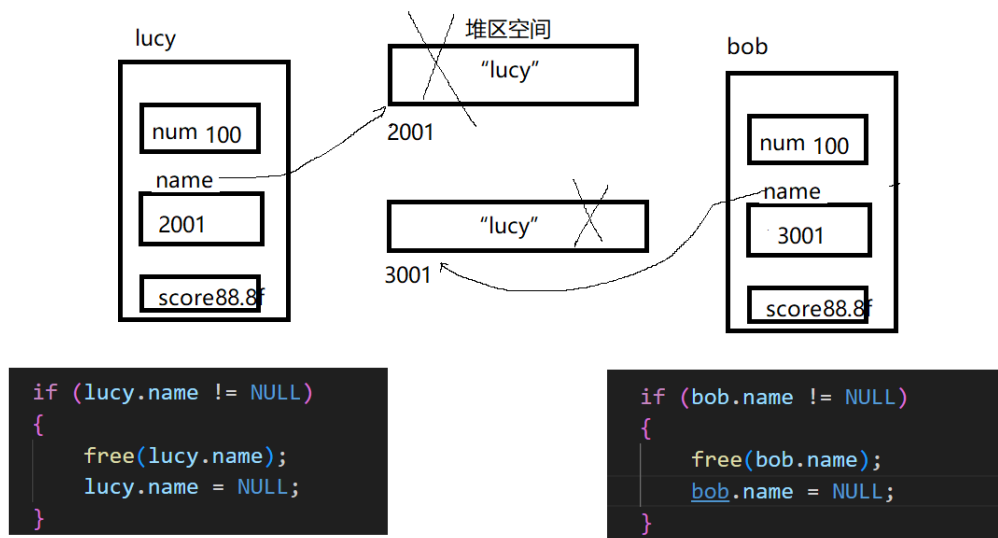
    lucy.name = NULL;
}

if (bob.name != NULL)
{
    free(bob.name);

    bob.name = NULL;
}
}

```

2、深拷贝：让结构体变量的指针成员 指向独立的堆区



```

void test02()

```

```

{

```

```
STU lucy;

lucy.num = 100;

lucy.name = (char *)malloc(32);

strcpy(lucy.name, "lucy");

lucy.score = 88.8f;


STU bob;

// 将非指针成员的内容直接赋值过去

bob = lucy;

// 让指针成员指向独立的堆区空间

bob.name = (char *)malloc(32);

strcpy(bob.name, lucy.name);


printf("%d %s %f\n", bob.num, bob.name, bob.score);


if (lucy.name != NULL)
{
    free(lucy.name);

    lucy.name = NULL;
}


if (bob.name != NULL)
```



```

{
    free(bob.name);

    bob.name = NULL;
}

}

int main(int argc, char const *argv[])
{
    test02();

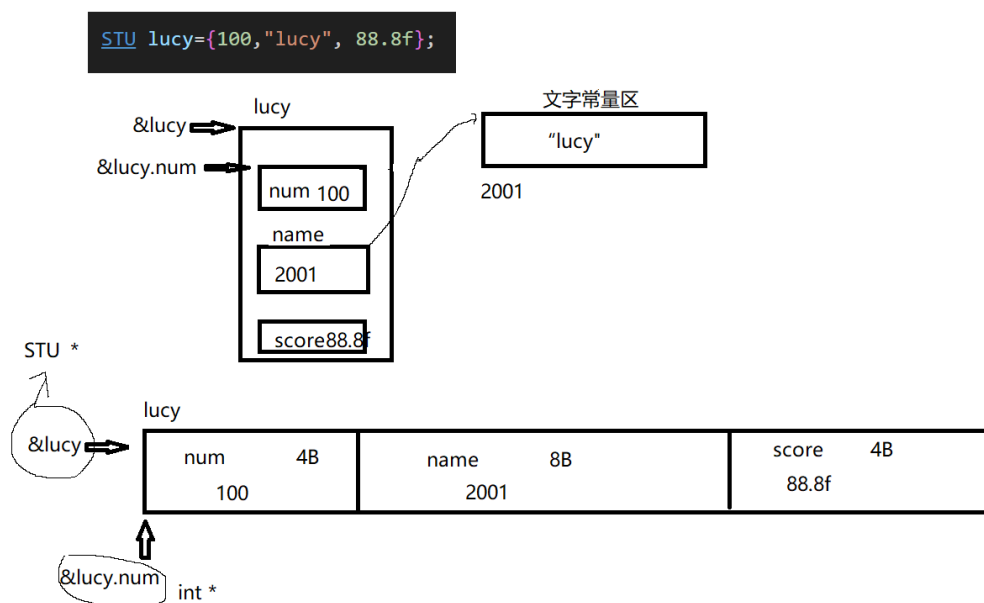
    return 0;
}

```

知识点 5 【结构体指针变量】

结构体指针变量 保存的是结构体变量地址。

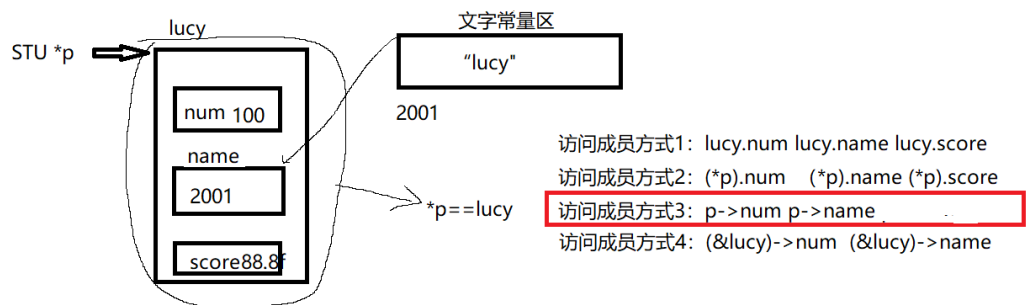
1、结构体变量的地址



2、结构体指针变量

```
STU lucy={100,"lucy", 88.8f};
```

```
STU *p = &lucy;
```



```
51 void test03()  
52 {  
53     STU lucy = {100, "lucy", 88.8f};  
54  
55     // 结构体指针变量p  
56     STU *p = &lucy;  
57  
58     printf("%d %s %f\n", p->num, p->name, p->score);  
59 }  
60 int main(int argc, char const *argv[])  
61 {  
62     test03();  
63     return 0;  
64 }  
65
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
edu@edu:~/work/c/day08$ gcc 01_code.c  
edu@edu:~/work/c/day08$ ./a.out  
100 lucy 88.800003  
edu@edu:~/work/c/day08$
```

```

51 void test03()
52 {
53     STU lucy;
54     STU *p = &lucy;
55
56     p->name = (char *)malloc(32);
57     printf("请输入num name score:");
58     scanf("%d %s %f", &p->num, p->name, &p->score);
59
60     printf("%d %s %f\n", p->num, p->name, p->score);
61     free(p->name);
62 }

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```

● edu@edu:~/work/c/day08$ gcc 01_code.c
● edu@edu:~/work/c/day08$ ./a.out
  请输入num name score:10 lucy 88
10 lucy 88.000000
○ edu@edu:~/work/c/day08$ 

```

知识点 6 【结构体指针的拓展】(建议多分析)

```

typedef struct stu
{
    int num;

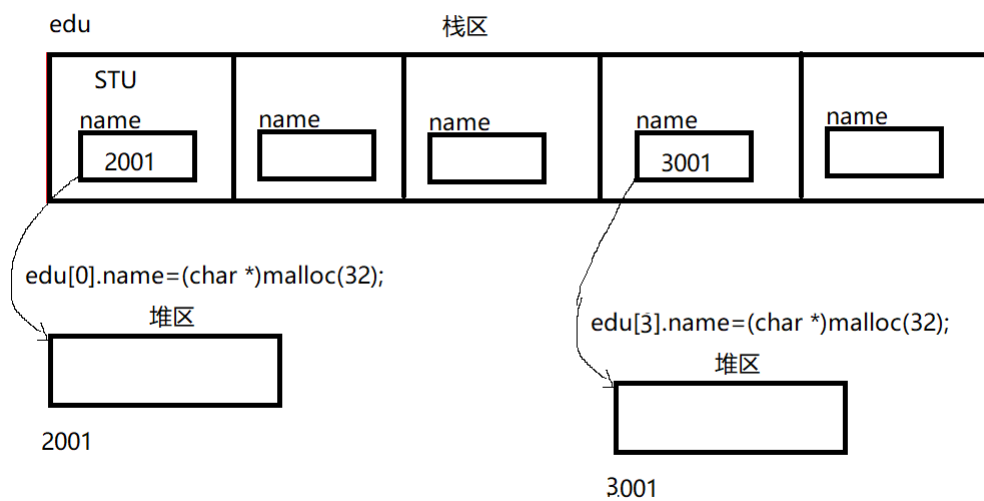
    char *name; // 指针成员 保存的是外部字符串的首元素地址

    float score;
} STU; // STU 就是 struct stu 类型

```

1、结构体数组在栈区 结构体指针成员指向堆区。

STU edu[5]



```
void test04()
{
    //结构体数组在 栈区

    STU edu[5];

    int n = sizeof(edu)/sizeof(edu[0]);

    //为数组的每个元素中的 name 申请堆区

    int i=0;

    for ( i = 0; i < n; i++)
    {

        edu[i].name = (char *)malloc(32);

        if(NULL == edu[i].name)

            return;
    }
}
```

```
}
```

```
//获取键盘输入
```

```
for ( i = 0; i < n; i++)
```

```
{
```

```
    scanf("%d %s %f", &edu[i].num, edu[i].name, &edu[i].score);
```

```
}
```

```
//输出
```

```
for ( i = 0; i < n; i++)
```

```
{
```

```
    printf("%d %s %f\n", edu[i].num, edu[i].name, edu[i].score);
```

```
}
```

```
//释放空间
```

```
for ( i = 0; i < n; i++)
```

```
{
```

```
    if(edu[i].name != NULL)
```

```
    {
```

```
        free(edu[i].name);
```

```
        edu[i].name=NULL;
```

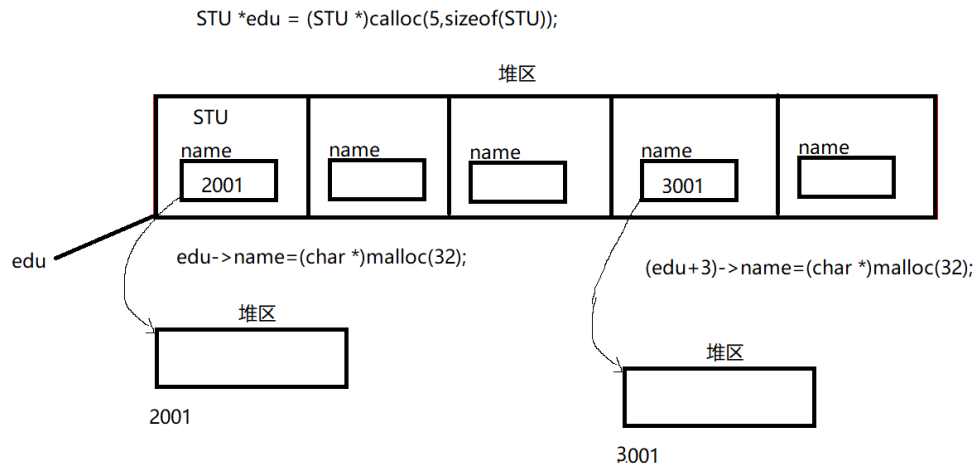
```
    }
```

```

    }
}

```

2、结构体数组在堆区 指针成员指向堆区。



```

void test05()
{
    // 结构体数组在 栈区

    STU *edu = (STU *)calloc(5, sizeof(STU));

    if (NULL == edu)
    {
        return;
    }

    int n = 3;

    // 为数组的每个元素中的 name 申请堆区

```

```
int i = 0;

for (i = 0; i < n; i++)

{

    (edu + i)->name = (char *)malloc(32);

    if (NULL == (edu + i)->name)

        return;

}


// 获取键盘输入

for (i = 0; i < n; i++)

{

    scanf("%d %s %f", &(edu + i)->num, (edu + i)->name, &(edu + i)->score);

}


// 输出

for (i = 0; i < n; i++)

{

    printf("%d %s %f\n", (edu + i)->num, (edu + i)->name, (edu + i)->score);

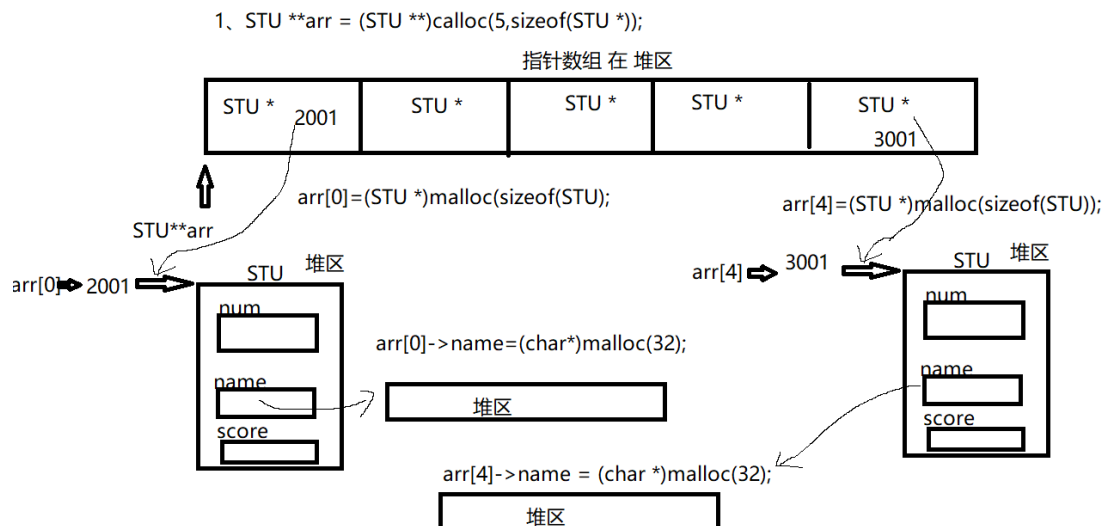
}


// 释放指针成员 指向的空间

for (i = 0; i < n; i++)
```

```
{  
  
    if ((edu + i)->name != NULL)  
  
    {  
  
        free((edu + i)->name);  
  
        (edu + i)->name = NULL;  
  
    }  
  
}  
  
// 释放结构体数组的空间（堆区）  
  
if (edu != NULL)  
  
{  
  
    free(edu);  
  
    edu = NULL;  
  
}  
  
}
```


3、结构体 STU 指针数组在堆区 每个元素指向堆区 元素的指针成员指向堆区。



```
void test06()
{
    //为指针数组申请堆区空间

    int n = 5;

    STU **arr=(STU **)calloc(n,sizeof(STU *));

    if(NULL == arr)

    {

        printf("calloc\n");

        return;

    }

    //为指针数组的每个元素申请空间
```

```
int i=0;

for ( i = 0; i < n; i++)

{

    //arr[i]就是指针数组的每个元素为 STU *

    arr[i] = (STU *)malloc(sizeof(STU));

    if(arr[i] == NULL)

        return;

    //为每个元素中的指针成员 name 申请空间

    arr[i]->name = (char *)malloc(32);

    if(arr[i]->name == NULL)

        return;

}

//获取键盘输入

// 获取键盘输入

for (i = 0; i < n; i++)

{

    scanf("%d %s %f", &arr[i]->num, arr[i]->name, &arr[i]->score);

}

// 输出
```

```
for (i = 0; i < n; i++)  
  
{  
  
    printf("%d %s %f\n", arr[i]->num, arr[i]->name, arr[i]->score);  
  
}
```

```
for ( i = 0; i < n; i++)  
  
{  
  
    //释放指针数组中每个元素指向的结构体中指针成员指向的堆区空间  
  
    if(arr[i]->name != NULL)  
  
    {  
  
        free(arr[i]->name);  
  
        arr[i]->name=NULL;  
  
    }  
  
  
    //释放指针数组每个元素指向的堆区空间  
  
    if(arr[i] != NULL)  
  
    {  
  
        free(arr[i]);  
  
        arr[i]=NULL;  
  
    }  
  
}
```

```
//释放指针数组的空间

if(arr!= NULL)

{

    free(arr);

    arr=NULL;

}

}
```

知识点 7 【结构体**指针成员**为函数指针】

```
int my_add(int x, int y)

{

    return x + y;

}

int my_sub(int x, int y)

{

    return x - y;

}

struct calc

{

    int x;
```

```

int y;

// 函数指针成员

int (*func)(int, int);

};

void test07()
{

    struct calc ob1 = {100, 200, my_add};

    printf("%d\n", ob1.func(ob1.x, ob1.y));

    struct calc ob2 = {100, 200, my_sub};

    printf("%d\n", ob2.func(ob2.x, ob2.y));

}

```

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

● edu@edu:~/work/c/day08$ gcc 01_code.c
● edu@edu:~/work/c/day08$ ./a.out
300
-100
○ edu@edu:~/work/c/day08$ 

```

知识点 8 【结构体对齐规则--默认规则】

1、知识点的引入

```
struct A
```

```

{

    char a; // 1B

    int b; // 4B

};

void test01()

{

    //struct A 的大小为啥不是 5 而是 8B

    printf("%ld\n", sizeof(struct A)); // 8B

}

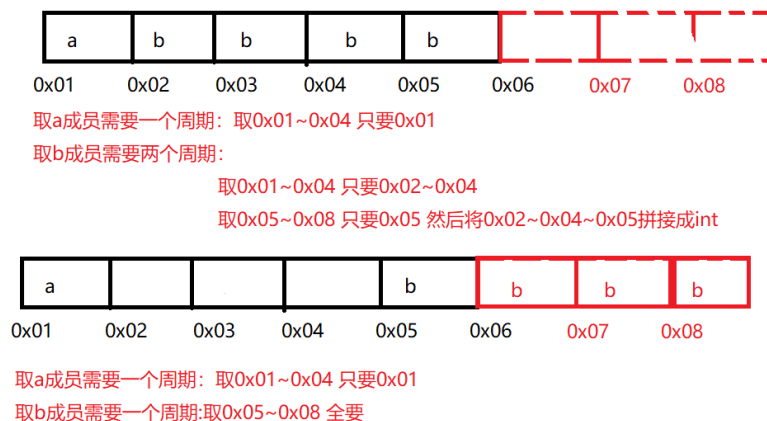
```

```

struct A
{
    char a; // 1B
    int b; // 4B
};

```

假如CPU的提取单位为4字节:



2、结构体自动对齐规则的步骤（背）

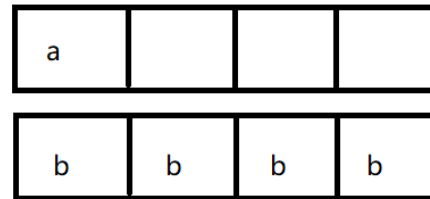
- 1、确定分配单位：一行分配多少字节（结构体中 最大的基本类型长度就是分配单位）
- 2、成员的偏移量：相对于结构体的起始位置的偏移字节数（成员自身大小的整数倍）
- 3、结构体总大小：分配单位的整数倍。

案例 1：

```

struct A
{
    char a; // 1B
    int b;   // 4B
};

```



假如CPU的提取单位为4字节:

案例 2:

```

1  #include <stdio.h>
2
3  struct A
4  {
5      char a; // 1B
6      int b;   // 4B
7      short c;
8  };
9  void test01()
10 {
11     struct A ob1;
12     printf("&ob1.a = %lu\n", &ob1.a);
13     printf("&ob1.b = %lu\n", &ob1.b);
14     printf("&ob1.c = %lu\n", &ob1.c);
15 }
16
17 int main(int argc, char const *argv[])
18 {

```

96 →

100 →

104 →

```

19 }
20
21 int main(int argc, char const *argv[])
22 {
23     test01();
24     return 0;
25 }

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS +

```

edu@edu:~/work/c/day08$ ./a.out
&ob1.a = 140727741837696
&ob1.b = 140727741837700
&ob1.c = 140727741837704
edu@edu:~/work/c/day08$

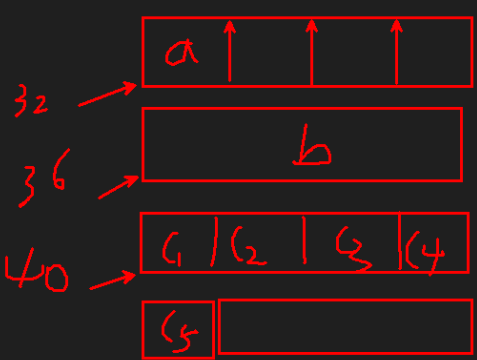
```

案例 3:

```

1  #include <stdio.h>
2
3  struct A
4  {
5      char a; // 1B
6      int b;  // 4B
7      char c[5];
8  };
9  void test01()
10 {
11     struct A ob1;
12     printf("%ld\n", sizeof(struct A));
13     printf("&ob1.a = %lu\n", &ob1.a);
14     printf("&ob1.b = %lu\n", &ob1.b);
15     printf("&ob1.c = %lu\n", ob1.c);
16 }
17 int main(int argc, char const *argv[])
18 {

```



```

16
&ob1.a = 140724228768032
&ob1.b = 140724228768036
&ob1.c = 140724228768040
edu@edu:~/work/c/day08$


```

3、结构体成员的顺序 可以决定空间大小

```

struct A
{
    char a; // 1B
    int b;  // 4B
    short c;
};

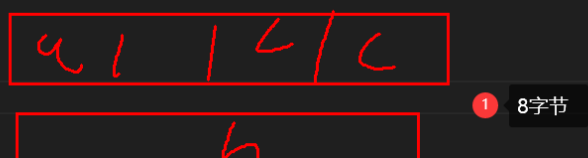
```



```

3  struct A
4  {
5      char a; // 1B
6      short c;
7      int b; // 4B
8  };

```



4、结构体嵌套结构体的对齐规则

- 1、确定分配单位：一行分配多少字节 == 所有结构体中最大的基本类型
- 2、成员的偏移量：相对于结构体的起始位置的偏移字节数（成员自身大小的整数倍）

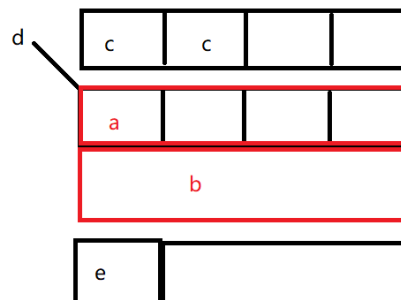
结构体成员的偏移量 == 该结构体最大基本类型的整数倍

结构体成员中的成员偏移量 == 相对于结构体成员的偏移量

3、结构体总大小：分配单位的整数倍。

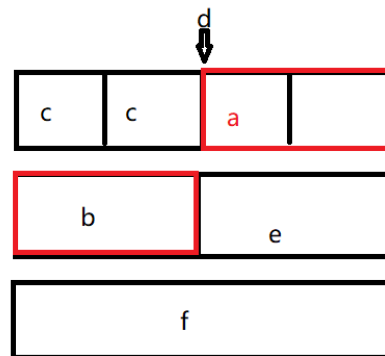
结构体成员的总大小 == 结构体成员中最大的基本类型整数倍

```
15
16 struct B
17 {
18     char a;
19     int b;
20 };
21 struct C
22 {
23     short c;
24     struct B d;
25     char e;
26 };
```



案例 1:

```
16 struct B
17 {
18     char a;
19     short b;
20 };
21 struct C
22 {
23     short c;
24     struct B d;
25     short e;
26     int f;
27 };
```



知识点 9 【结构体对齐规则--强制对齐】

#pragma pack (value)时的指定对齐值 value。

注意：1.value 只能是：1 2 4 8 等

1、确定分配单位：一行分配多少字节（ $\min(\text{value}, \text{结构体中最大的基本类型长度})$ ）

2、成员的偏移量：相对于结构体的起始位置的偏移字节数（成员自身大小的整数倍）

3、结构体总大小：分配单位的整数倍。

```
34  #pragma pack(1)
35  struct D
36  {
37      char a;
38      int b;
39  };
40  void test03()
41  {
42      printf("%ld\n", sizeof(struct D));
43  }
44  int main(int argc, char const *argv[])
45  {
46      test03();
47      return 0;
48  }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
● edu@edu:~/work/c/day08$ gcc 02_code.c
● edu@edu:~/work/c/day08$ ./a.out
5
○ edu@edu:~/work/c/day08$
```

```
33 #pragma pack(2)
34 struct D
35 {
36     char a;
37     int b;
38 };
39 void test03()
40 {
41     printf("%ld\n", sizeof(struct D));
42 }
43 int main(int argc, char const *argv[])
44 {
45     test03();
46     return 0;
47 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

edu@edu:~/work/c/day08\$ gcc 02_code.c

edu@edu:~/work/c/day08\$./a.out

6

edu@edu:~/work/c/day08\$

```
33 #pragma pack(8)
34 struct D
35 {
36     char a;
37     int b;
38     int c;
39 };
40 void test03()
41 {
42     printf("%ld\n", sizeof(struct D));
43 }
44 int main(int argc, char const *argv[])
45 {
46     test03();
47     return 0;
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

Open file in editor (ctrl + click)

```
edu@edu:~/work/c/day08$
edu@edu:~/work/c/day08$ ./a.out
12
edu@edu:~/work/c/day08$
```

```
33 // #pragma pack(8)
34 struct D
35 {
36     char a;
37     double b; // 在ubuntu64位的gcc环境下 double按8字节对齐
38 };
39 void test03()
40 {
41     printf("%ld\n", sizeof(struct D));
42 }
43 int main(int argc, char const *argv[])
44 {
45     test03();
46     return 0;
47 }
48
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
edu@edu:~/work/c/day08$ gcc 02_code.c
edu@edu:~/work/c/day08$ ./a.out
16
edu@edu:~/work/c/day08$
```

知识点 10 【结构体的位域】

1、位域的概述

结构体中成员所在空间以二进制位计算 称之为位域或位段。

```
struct A
{
    // a 所占的位数 不能超过 自身类型的二进制位数

    unsigned int a : 3; // 位域 a 类型为 unsigned int 只是占空间 3 位二进制位
};
```

2、给位域赋值时 注意不要超过它所占位的最大值

```
struct A
{
```

```

// 0~7

unsigned int a : 3; // 位域 a 类型为 unsigned int 只是占空间 3 位二进制位
};

void test01()
{

    struct A ob1;

    ob1.a = 10; // 1010

    printf("%u\n", ob1.a); // 2
}

```

```

14 struct B
15 {
16     int a : 3;
17 };
18 void test02()
19 {
20     struct B ob1;
21     ob1.a = 14;           // 1110
22     printf("%d\n", ob1.a); // -2
23 }
24 int main(int argc, char const *argv[])
25 {
26     test02();
27     return 0;

```

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

PORTS

```

edu@edu:~/work/c/day08$ ./a.out
-2
edu@edu:~/work/c/day08$

```

3、相邻位域可以压缩

```
25 struct C
26 {
27     // 相邻且相同类型的位域 为相邻位域
28     // 相邻位域可以压缩 但是压缩的位数 不能超过 成员自身类型为位数
29     unsigned char a : 4;
30     unsigned char b : 2;
31     unsigned char c : 3;
32 };
```

4、直接另一个存储单元

```
25 struct C
26 {
27     // 相邻且相同类型的位域 为相邻位域
28     // 相邻位域可以压缩 但是压缩的位数 不能超过 成员自身类型为位数
29     unsigned char a : 4;
30     unsigned char : 0; // 另起一个存储单元
31     unsigned char b : 2;
32 };
```

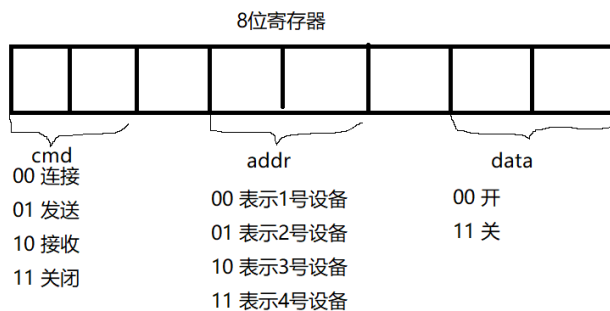
大小2字节

5、无意义位宽

```
25 struct C
26 {
27     // 相邻且相同类型的位域 为相邻位域
28     // 相邻位域可以压缩 但是压缩的位数 不能超过 成员自身类型为位数
29     unsigned char a : 4;
30     unsigned char : 2; // 另起一个存储单元
31     unsigned char b : 2;
32 };
```



6、位域的综合应用



```
struct REG
{
    unsigned char cmd:2;
    unsigned char :1;
    unsigned char addr:2;
    unsigned char :1;
    unsigned char data:2;
};
```

案例1: 给3号设备 发送 开指令

```
struct REG ob1;
ob1.cmd=1;
ob1.addr=2;
ob1.data=0;
```

知识点 11 【共有体】

共用体 关键字为 union，它的所有成员共享同一块空间。

```
union D
{
    char a;

    short b;

    int c;
};
```

a b c 共享同一块空间 空间的大小由最大的成员大小决定。

```
39 union D
40 {
41     char a;
42     short b;
43     int c;
44 };
45
46 void test04()
47 {
48     printf("%ld\n", sizeof(union D));
49 }
50 int main(int argc, char const *argv[])
51 {
52     test04();
53     return 0;
54 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
● edu@edu:~/work/c/day08$ ./a.out
4
○ edu@edu:~/work/c/day08$
```

案例 1:


```

39  union D
40  {
41      char a;
42      short b;
43      int c;
44  };
45
46  void test04()
47  {
48      printf("%ld\n", sizeof(union D));
49      union D ob1;
50      ob1.a = 10;
51      ob1.b = 20;
52      ob1.c = 30;
53      printf("%d\n", ob1.a+ob1.b+ob1.c); //90
54

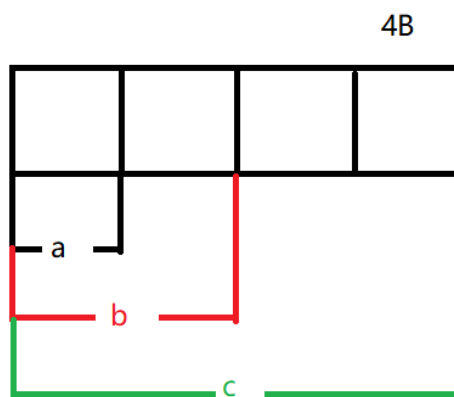
```

共用体 所有的成员共享同一块空间 空间的大小由最大的成员类型大小决定，成员操作空间大小 由自身类型大小决定（重要）

```

39  union D
40  {
41      char a;
42      short b;
43      int c;
44  };
45

```



```

union D ob1;

ob1.c=0x01020304;

ob1.b=0x0102;

ob1.a=0x01;

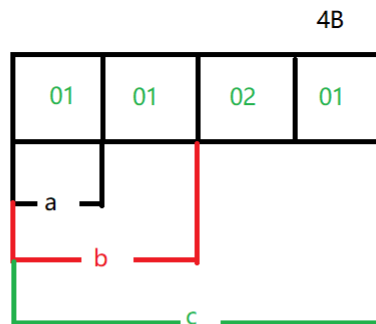
printf("%x\n", ob1.a+ob1.b+ob1.c);

```

```

39 union D
40 {
41     char a;
42     short b;
43     int c;
44 };

```



```

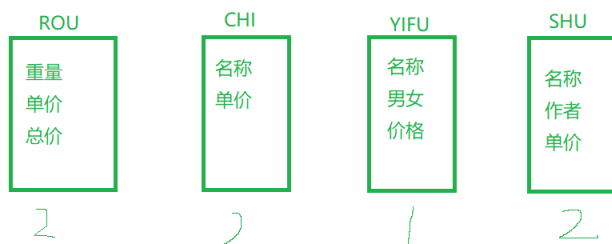
union D ob1;
ob1.c=0x01020304;
ob1.b=0x0102;
ob1.a=0x01;
printf("%x\n", ob1.a+ob1.b+ob1.c);

```

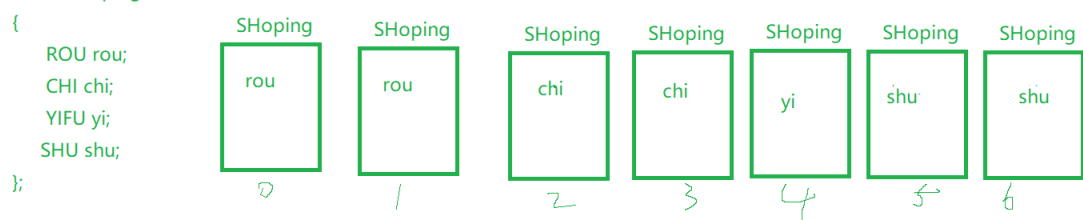
↓
 0x01
 ↓
 0x0101
 ↓
 0x01020101

0x01020101
 0x0101

 0x01
 0x01020203



```
union Shopping
```



知识点 13【枚举】

枚举：将变量将要赋的值——列举出来。关键字为 enum

```

55 // 枚举类型 枚举列表中的值 默认从0开始递增
56 enum POOKER{HONGTAO,MEIHUA,FANGKUAI,HEITAO};
57 void test05()
58 {
59     enum POOKER pooker_color = HONGTAO; // 赋值其他值比如30 没有任何意义
60     printf("%d %d %d %d\n", HONGTAO, MEIHUA, FANGKUAI, HEITAO); //0 1 2 3
61 }

```

```

55 // 枚举类型 枚举列表中的值 默认从0开始递增
56 enum POOKER{HONGTAO,MEIHUA=3,FANGKUAI,HEITAO};
57 void test05()
58 {
59     enum POOKER pooker_color = HONGTAO; // 赋值其他值比如30 没有任何意义
60     printf("%d %d %d %d\n", HONGTAO, MEIHUA, FANGKUAI, HEITAO); //0 3 4 5
61 }

```

案例 1: c 语言不支持 bool 可以使用枚举实现

```

69 enum BOOL{false,true};
70 void test06()
71 {
72     enum BOOL flag;
73     flag = true;
74     if(flag)
75     {
76         printf("为真\n");
77     }
78     else{
79         printf("为假\n");
80     }
81 }
82 int main(int argc, char const *argv[])
83 {

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```

● edu@edu:~/work/c/day08$ ./a.out
为真
○ edu@edu:~/work/c/day08$

```