1、书接上回 --> 结构体

1.1 定义结构体指针数组

```
1. 结构体指针数组:本质是一个数组，数组的每个成员都是一个结构体指针类型的地址

2. 定义结构体指针数组
    struct 结构体名 * 结构体指针数组名[元素个数];

3. 结构体指针数组的初始化
    struct 结构体名 * 结构体指针数组名[元素个数] =
            {&普通结构体变量0, &普通结构体变量1, .... }

4. 结构体指针数组中每个成员的访问
    结构体指针数组名[下标]->成员变量名;
```

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

// 1. 声明一个结构体类型
struct Teacher {
    char name[20];
    int age;
    float salary;
};

```

```c
void print_info(struct Teacher *p[], int len)
// void print_info(struct Teacher **p, int len)
{
    for (int i = 0; i < len; i++)
    {
        printf("姓名：%s\t", p[i]->name);
        printf("年龄：%d\t", p[i]->age);
        printf("薪资：%f\n", p[i]->salary);
    }
    for (int i = 0; i < len; i++)
    {
        printf("姓名：%s\t", (*p[i]).name);
        printf("年龄：%d\t", (*p[i]).age);
        printf("薪资：%f\n", (*p[i]).salary);
    }
}
int main(int argc, const char *argv[])
{
    /*your code*/
    // 定义结构体变量
    struct  Teacher tea1 = {"xiaozhou", 18, 20000},
    tea2 = {
        .name = "xiaodai",
        .age = 18,
        .salary = 25000
    };

    // 2. 定义一个结构体指针数组
    struct Teacher * p_array[2] = {&tea1, &tea2};

    print_info(p_array, 2);
    return 0;
}
```

1.2 定义结构体数组指针

```
1. 结构体数组指针 :本质是一个指针，指向的是一个二维数组

2. 定义结构体数组指针
    struct 结构体名 (*结构体数组指针变量名)[列宽];

3. 结构体数组指针的初始化
    struct 结构体名 二维数组名[行宽][列宽] = {};
    struct 结构体名 (*结构体数组指针变量名)[列宽] = 二维数组名;

4. 结构体数组指针指向的二维数组的成员的访问
    4.1> 将结构体数组指针当成二维数组名使用
        结构体数组指针变量名[行下标][列下标].成员变量名;

    4.2> 采用地址加偏移量的方式
        (*(结构体数组指针变量名 + 行偏移) + 列偏移)->成员变量名;

        (*(*(结构体数组指针变量名 + 行偏移) + 列偏移)).成员变量名;
```

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

// 1. 声明一个结构体类型
struct Teacher {
    char name[20];
    int age;
    float salary;
};

void print_info(struct Teacher (*arr_p)[2], int row, int col)
{
#if  0
    for (int i = 0; i < row ; i++)
    {
        for (int j = 0; j < col ; j++)
        {
            // 1. 将数组指针变量名当成二维数组名使用即可
            printf("姓名：%s\t", arr_p[i][j].name);
            printf("年龄：%d\t", arr_p[i][j].age);
            printf("薪资：%f\n", arr_p[i][j].salary);
        }
    }
#endif
#if  0
    for (int i = 0; i < row ; i++)
    {
        for (int j = 0; j < col ; j++)
        {
            // 1. 利用地址偏移的方式
            printf("姓名：%s\t", (*(*(arr_p + i) + j)).name);
            printf("年龄：%d\t", (*(*(arr_p + i) + j)).age);
            printf("薪资：%f\n", (*(*(arr_p + i) + j)).salary);
        }
    }
#endif
    for (int i = 0; i < row ; i++)
    {
        for (int j = 0; j < col ; j++)
        {
            // 1. 利用地址偏移的方式
            printf("姓名：%s\t", (*(arr_p + i) + j)->name);
            printf("年龄：%d\t", (*(arr_p + i) + j)->age);
            printf("薪资：%f\n", (*(arr_p + i) + j)->salary);
        }
    }
}
int main(int argc, const char *argv[])
{
    /*your code*/
    // 2. 定义一个结构体一维数组
    struct Teacher tea_arr[2] = {
        [0] = {
```

```
            .name = "xiaozhang",
            .age = 20,
            .salary = 15000,
        },
        [1] = {
            .name = "xiaoli",
            .age = 22,
            .salary = 16000,
        },
    };

    // 3. 定义结构体数组指针,指向2列的二维数组
    struct Teacher (*array_p)[2] = &tea_arr;
    print_info(array_p, 1, 2);

    // 4. 定义一个2行2列的二维数组
     struct Teacher tea_arr2[2][2] =
     {
        [0][0] = {
            .name = "xiaoli",
            .age = 22,
            .salary = 16000,
        },
        [0][1] = {
            .name = "xiaosun",
            .age = 23,
            .salary = 17000,
        },
        [1][0] = {
            .name = "xiaowu",
            .age = 24,
            .salary = 17000,
        },
        [1][1] = {
            .name = "xiaoqiao",
            .age = 25,
            .salary = 18000,
        },
     };
    array_p = tea_arr2;
    print_info(array_p, 2, 2);
    return 0;
}
```

1.3 结构体中的成员包含指针类型的成员

```
struct 结构体名 {
    数据类型  *指针变量名;   // 指向普通的变量，指向堆区空间
};
```

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

```c
#include <stdlib.h>

// 1. 声明结构体类型
struct Person{
    char *name;
    char sex;
    int age;
};

void print_info(struct Person *per)
{
    printf("姓名：%s\t性别：%c\t年龄：%d\n",
            per->name,per->sex,per->age);
}
int main(int argc, const char *argv[])
{
    /*your code*/
    // 2. 定义结构体变量，让结构体变量中的指针成员指向字符数组。
    char name[20] = "xiaozhao";
    struct Person per1 = {name, 'M', 18};
    print_info(&per1);

    // 3. 定义结构体变量，让结构体变量中的指针成员指向堆区空间
    struct Person per2;
    per2.name = (char *)malloc(sizeof(char)*20);
    strcpy(per2.name ,"xiaoqian");
    per2.age = 19;
    per2.sex = 'W';
    print_info(&per2);

    free(per2.name);
    per2.name = NULL;

    // 4. 定义结构体指针变量，在堆区分配空间，
    // 结构体指针变量指向的空间的指针类型的成员也在堆区分配空间。

    struct Person *per_p =
            (struct Person *)malloc(sizeof(struct Person));
    per_p->name = (char *)malloc(sizeof(char)*20);

    strcpy(per_p->name, "xiaoli");
    per_p->sex = 'M';
    per_p->age = 20;
    print_info(per_p);

    free(per_p->name);
    per_p->name = NULL;
    free(per_p);
    per_p = NULL;

    return 0;
}
```

1.4 结构体中的成员包含函数指针类型的成员

```
struct 结构体名{
    返回类型 (*函数指针变量名)(参数列表);
};
```

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

struct Complex {
    int m_i;
    int m_j;
};

struct Cal{
    struct Complex (*func_p)(struct Complex l, struct Complex r);
};

struct Complex add_func(struct Complex l, struct Complex r)
{
    struct Complex sum;
    sum.m_i = l.m_i + r.m_i;
    sum.m_j = l.m_j + r.m_j;
    return sum;
}
int main(int argc, const char *argv[])
{
    /*your code*/
    // 定义结构体变量
    struct Complex c1 = {10,20}, c2 = {30, 40}, sum;
    struct Cal cal;
    // 对结构体中的成员函数指针变量进行初始化
    cal.func_p = add_func;

    // 通过结构体变量得到函数指针成员，整体当成函数名使用即可。
    sum = cal.func_p(c1, c2);
    printf("%d + %di\n", sum.m_i, sum.m_j);
    return 0;
}
```

1.5 typedef定义新的结构体类型

```
typedef struct /*结构体名 (可以省略不写)*/ {
    结构体成员;
} 结构体类型名;

此时"结构体类型名"就是使用typedef重新定义的新的结构体类型，
等价于“struct 结构体名”，可以使用“结构体类型名”定义结构体变量。

```

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
// 使用typedef对结构类型起别名，重新定义新的结构体类型
typedef struct /*Complex*/ {
    int m_i;
    int m_j;
}Complex_t;

typedef struct /*Cal*/ {
    Complex_t (*func_p)(Complex_t l, Complex_t r);
}Cal_t;

Complex_t add_func(Complex_t l, Complex_t r)
{
    Complex_t sum;
    sum.m_i = l.m_i + r.m_i;
    sum.m_j = l.m_j + r.m_j;
    return sum;
}
int main(int argc, const char *argv[])
{
    /*your code*/
    // 定义结构体变量
    Complex_t c1 = {10,20}, c2 = {30, 40}, sum;
    Cal_t cal;
    // 对结构体中的成员函数指针变量进行初始化
    cal.func_p = add_func;

    // 通过结构体变量得到函数指针成员，整体当成函数名使用即可。
    sum = cal.func_p(c1, c2);
    printf("%d + %di\n", sum.m_i, sum.m_j);
    return 0;
}
```

```c
// 声明结构体类型
typedef struct Person {
    char *name;
    char sex;
    int age;
}Person_t;

// 结构体中嵌套一个结构体类型的指针变量。
typedef struct Student {
    struct Person *per;
    float score;
}Student_t;
```

使用“Student_t”定义结构体指针变量，在堆区分配空间，
    然后对per成员在堆区分配空间，并初始化，
    在对name成员在堆区分配空间，并初始化。

```
```

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

// 声明结构体类型
typedef struct Person {
    char *name;
    char sex;
    int age;
}Person_t;

// 结构体中嵌套一个结构体类型的指针变量。
typedef struct Student {
    struct Person *per;
    float score;
}Student_t;

int main(int argc, const char *argv[])
{
    /*your code*/
    // 1. 定义结构体指针类型的变量，并在堆区分配空间
    Student_t  *stu_p = NULL;
    stu_p = (Student_t *)malloc(sizeof(Student_t));

    // 对stu_p->per成员分配堆区空间
    stu_p->per = (Person_t *) malloc(sizeof(Person_t));

    // 对stu_p->per->name成员分配堆区空间
    stu_p->per->name = (char *)malloc(sizeof(char)*20);

    // 2. 对结构体指针变量指向的堆区空间进行初始化
    strcpy(stu_p->per->name, "zhangsan");
    stu_p->per->sex = 'M';
    stu_p->per->age = 18;
    stu_p->score = 99;

    // 3. 打印结果
    printf("%s %c %d %f\n", stu_p->per->name,
            stu_p->per->sex, stu_p->per->age,
            stu_p->score);

    // 4. 释放堆区的空间
    free(stu_p->per->name);
    stu_p->per->name = NULL;
    free(stu_p->per);
    stu_p->per = NULL;
    free(stu_p);
    stu_p = NULL;
    return 0;
}
```

1.6 结构体的内存对齐

1.6.1 32位操作系统的结构体内存对齐

```
1. 关于结构体中的成员的内存对齐，只考虑基本的数据类型，不考虑构造类型，
    构造类型也是由基本类型构成。

2. 声明结构体类型的大小
    1> 如果结构体中的成员最大的成员占1个字节的空间，
        则结构体的大小为1的整数倍
    2> 如果结构体中的成员最大的成员占2个字节的空间，
        则结构体的大小为2的整数倍
    3> 如果结构体中的成员最大的成员占4个或者8个字节的空间，
        则结构体的大小为4的整数倍

3. 结构体中成员的地址
    1> 如果结构体中的成员占1个字节的空间，则此成员的地址为1的整数倍
        char / unsigned char
    2> 如果结构体中的成员占2个字节的空间，则此成员的地址为2的整数倍
        short / unsigned short
    3> 如果结构体中的成员占4个或者8个字节的空间，则此成员的地址为4的整数倍
        int / unsigned int / long int / float / 指针 / 函数指针
        double / long long int

4. 将程序编译生成32位的可执行程序
    gcc ****.c  -m32
```

```c
#include <stdio.h>

typedef struct {
    char a[5];
    char b;
    char c;
}A_t;

typedef struct {
    char a[5];      // 5字节
                    // 保留1字节
    short b;        // 2字节
    char c;         // 1字节
                    // 保留1字节
}B_t;

typedef struct {
    char a[5];  // 5字节
                // 保留1字节
    short b;    // 2字节
    char c;     // 1字节
                // 保留3字节
    int d;      // 4字节
    int *p;     // 4字节
}C_t;

typedef struct {
    char a[5];      // 5字节
                    // 保留3字节
```

```c
    int b;          // 4字节
    char c;         // 1字节
                    // 保留3字节
    long long int d;// 8字节
}D_t;
typedef struct {
    char a[5];      // 5字节
                    // 保留3字节
    int b;          // 4字节
    long long int c;// 8字节
    char d;         // 1字节
                    // 保留3字节
}E_t;

int main(int argc, const char *argv[])
{
    /*your code*/
    // 1> 如果结构体中的成员最大的成员占1个字节的空间，
    //  则结构体的大小为1的整数倍
    // 1> 如果结构体中的成员占1个字节的空间，则此成员的地址为1的整数倍
    //  char / unsigned char

    printf("A_t size = %d\n", sizeof(A_t));

    // 2> 如果结构体中的成员最大的成员占2个字节的空间，
    //  则结构体的大小为2的整数倍
    // 2> 如果结构体中的成员占2个字节的空间，则此成员的地址为2的整数倍
    //  short / unsigned short
    printf("B_t size = %d\n", sizeof(B_t));
    B_t  B;
    printf("&B_t.a address = %p\n", &B.a);
    printf("&B_t.b address = %p\n", &B.b);
    printf("&B_t.c address = %p\n", &B.c);

    // 3> 如果结构体中的成员最大的成员占4个或者8个字节的空间，
    //  则结构体的大小为4的整数倍
    // 3> 如果结构体中的成员占4个或者8个字节的空间，则此成员的地址为4的整数倍
    //  int / unsigned int / long int / float / 指针 / 函数指针
    //  double / long long int
    printf("C_t size = %d\n", sizeof(C_t));
    C_t  C;
    printf("&C_t.a address = %p\n", &C.a);
    printf("&C_t.b address = %p\n", &C.b);
    printf("&C_t.c address = %p\n", &C.c);
    printf("&C_t.d address = %p\n", &C.d);
    printf("&C_t.p address = %p\n", &C.p);

    printf("D_t size = %d\n", sizeof(D_t));
    D_t  D;
    printf("&D_t.a address = %p\n", &D.a);
    printf("&D_t.b address = %p\n", &D.b);
    printf("&D_t.c address = %p\n", &D.c);
    printf("&D_t.d address = %p\n", &D.d);

    printf("E_t size = %d\n", sizeof(E_t));
```

```
    E_t  E;
    printf("&E_t.a address = %p\n", &E.a);
    printf("&E_t.b address = %p\n", &E.b);
    printf("&E_t.c address = %p\n", &E.c);
    printf("&E_t.d address = %p\n", &E.d);
    return 0;
}
```

1.6.2 64位操作系统的结构体内存对齐

```
1. 关于结构体中的成员的内存对齐，只考虑基本的数据类型，不考虑构造类型，
    构造类型也是由基本类型构成。

2. 声明结构体类型的大小
    1> 如果结构体中的成员最大的成员占1个字节的空间，
        则结构体的大小为1的整数倍
    2> 如果结构体中的成员最大的成员占2个字节的空间，
        则结构体的大小为2的整数倍
    3> 如果结构体中的成员最大的成员占4个字节的空间，
        则结构体的大小为4的整数倍
    4> 如果结构体中的成员最大的成员占8个字节的空间，
        则结构体的大小为8的整数倍

3. 结构体中成员的地址
    1> 如果结构体中的成员占1个字节的空间，则此成员的地址为1的整数倍
        char / unsigned char
    2> 如果结构体中的成员占2个字节的空间，则此成员的地址为2的整数倍
        short / unsigned short
    3> 如果结构体中的成员占4个字节的空间，则此成员的地址为4的整数倍
        int / unsigned int  / float
    4> 如果结构体中的成员占8个字节的空间，则此成员的地址为8的整数倍
        double / long long int / long int / 指针 / 函数指针

```

```
#include <stdio.h>

typedef struct {
    char a[5];
    char b;
    char c;
}A_t;

typedef struct {
    char a[5];      // 5字节
                    // 保留1字节
    short b;        // 2字节
    char c;         // 1字节
                    // 保留1字节
}B_t;

typedef struct {
    char a[5];  // 5字节
```

```
                // 保留1字节
    short b;    // 2字节
    char c;     // 1字节
                // 保留3字节
    int d;      // 4字节
    int *p;     // 8字节
}C_t;

typedef struct {
    char a[5];      // 5字节
                    // 保留3字节
    int b;          // 4字节
    char c;         // 1字节
                    // 保留3字节
    long long int d;// 8字节
}D_t;
typedef struct {
    char a[5];      // 5字节
                    // 保留3字节
    int b;          // 4字节
                    // 保留4字节
    long long int c;// 8字节
    char d;         // 1字节
                    // 保留7字节
}E_t;

int main(int argc, const char *argv[])
{
    /*your code*/
    // 1> 如果结构体中的成员最大的成员占1个字节的空间，
    //  则结构体的大小为1的整数倍
    // 1> 如果结构体中的成员占1个字节的空间，则此成员的地址为1的整数倍
    //  char / unsigned char

    printf("A_t size = %ld\n", sizeof(A_t));

    // 2> 如果结构体中的成员最大的成员占2个字节的空间，
    //  则结构体的大小为2的整数倍
    // 2> 如果结构体中的成员占2个字节的空间，则此成员的地址为2的整数倍
    //  short / unsigned short
    printf("B_t size = %ld\n", sizeof(B_t));
    B_t  B;
    printf("&B_t.a address = %p\n", &B.a);
    printf("&B_t.b address = %p\n", &B.b);
    printf("&B_t.c address = %p\n", &B.c);

    // 3> 如果结构体中的成员最大的成员占4个字节的空间，
    //  则结构体的大小为4的整数倍
    // 3> 如果结构体中的成员占4个字节的空间，则此成员的地址为4的整数倍
    //  int / unsigned int / float
    printf("C_t size = %ld\n", sizeof(C_t));
    C_t  C;
    printf("&C_t.a address = %p\n", &C.a);
    printf("&C_t.b address = %p\n", &C.b);
    printf("&C_t.c address = %p\n", &C.c);
```

```
    printf("&C_t.d address = %p\n", &C.d);
    printf("&C_t.p address = %p\n", &C.p);

    // 4> 如果结构体中的成员最大的成员占8个字节的空间，
    //  则结构体的大小为8的整数倍
    // 4> 如果结构体中的成员占8个字节的空间，则此成员的地址为8的整数倍
    //  double / long long int / long int / 指针 / 函数指针
    printf("D_t size = %ld\n", sizeof(D_t));
    D_t  D;
    printf("&D_t.a address = %p\n", &D.a);
    printf("&D_t.b address = %p\n", &D.b);
    printf("&D_t.c address = %p\n", &D.c);
    printf("&D_t.d address = %p\n", &D.d);

    printf("E_t size = %ld\n", sizeof(E_t));
    E_t  E;
    printf("&E_t.a address = %p\n", &E.a);
    printf("&E_t.b address = %p\n", &E.b);
    printf("&E_t.c address = %p\n", &E.c);
    printf("&E_t.d address = %p\n", &E.d);
    return 0;
}
```

2、联合体/共用体 ---> union

2.1 联合体相关的介绍

```
1. 联合体概念
    联合体是一个构造类型，关键字union.
    联合体中可以包含很多不同类型的变量，联合中的所有的成员公用一块内存空间。

    联合体类型的大小为联合体中最大类型成员的整数倍，
    联合体也需要考虑内存对齐的问题。

2. 声明联合体类型
    typedef union /*联合体名(可以省略不写)*/ {
        数据类别 成员变量名1;
        数据类型 成员变量名2;
        .....
        数据类型 成员变量名n;
    } 联合体类型别名;

    可以使用“联合体类型别名”或者“union 联合体名”定义联合体类型的变量。
        联合体类型别名  变量名;
        union 联合体名 *指针变量名;

3. 定义联合体类型的变量，并进行初始化
        联合体类型别名  变量名;
        union 联合体名 *指针变量名 = &变量名;

        变量名.成员名 = 初始值;
        指针变量名->成员名 = 初始值;

4. 访问联合体类型变量中的成员
```

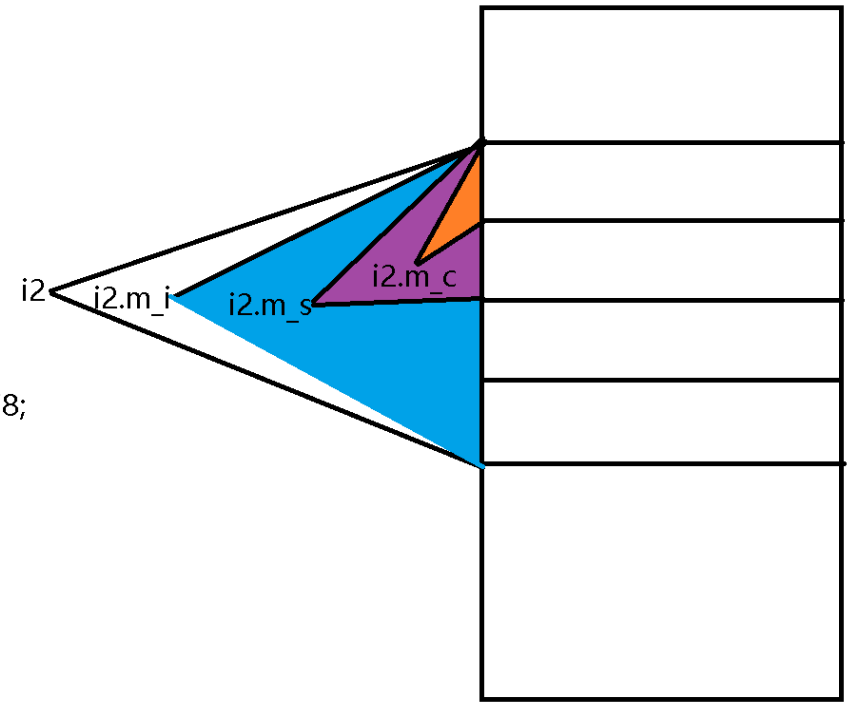
```
        变量名.成员名;
        指针变量名->成员名;

5. 在目前开发中，基本上不在使用联合体，联合体中的成员共用一块内存空间，
    相对来说如果使用不当，就是不安全。
    但是在内核的底层代码中依然存在着部分联合体的使用。
```

2.2 联合体的练习题

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

// 1. 声明联合体类型
typedef union {
    int m_i;
    short m_s;
    char m_c;
} Integer_t;        // 4字节

typedef union {
    int m_i;
    char c[5];
} data_t;       // 8字节

int main(int argc, const char *argv[])
{
    /*your code*/
    // 1. 验证联合体类型的大小为最大成员的整数倍
    printf("Integer_t size = %ld\n", sizeof(Integer_t));
    printf("data_t size = %ld\n", sizeof(data_t));

    // 2. 联合体类型中的成员公用同一块内存空间
    Integer_t  i1;
    printf("i1.m_i address = %p\n", &i1.m_i);
    printf("i1.m_s address = %p\n", &i1.m_s);
    printf("i1.m_c address = %p\n", &i1.m_c);

    // 3. 定义联合体类型的变量并对成员进行初始化，
    // 一般使用联合体的哪个成员就对那个成员进行初始化。
    Integer_t  i2;
    i2.m_i = 0x12345678;
    printf("i2.m_i = %#x\n", i2.m_i);    // 0x12345678
    printf("i2.m_s = %#x\n", i2.m_s);    // 0x5678
    printf("i2.m_c = %#x\n", i2.m_c);    // 0x78

    // 4. 定义联合类型的指针变量
    Integer_t * i_p = (Integer_t *)malloc(sizeof(Integer_t));
    i_p->m_i = 0x12345678;
    printf("i_p->m_i = %#x\n", i_p->m_i);    // 0x12345678
    printf("i_p->m_s = %#x\n", i_p->m_s);    // 0x5678
    printf("i_p->m_c = %#x\n", i_p->m_c);    // 0x78
    return 0;
```

```
}
```

```
typedef union {
    int m_i;
    short m_s;
    char m_c;
} Integer_t;

Integer_t  i2;
i2.m_i = 0x12345678;
```



练习题：

声明一个联合体类型，测试计算机的大小端存储的问题。

```c
typedef union little {
    unsigned int m_i;
    unsigned char m_c[4];
}little_t;
```

对联合体中的整型成员赋值，然后按照字节的方式进行访问即可。

```c
#include <stdio.h>
// 声明一个联合体类型
typedef union little {
    unsigned int m_i;
    unsigned char m_c[4];
}little_t;

int main(int argc, const char *argv[])
{
    /*your code*/
    // 定义联合体类型，并进行初始化
    little_t lit;
    lit.m_i = 0x12345678;
    // 大端对齐：低地址存放数据的高有效位，高地址存放数据的低有效位
    // 小端对齐：低地址存放数据的低有效位，高地址存放数据的高有效位
    for (int i = 0; i < 4; i++)
    {
        printf("&lit.m_c[%d] = %p = %#x\n",i, &lit.m_c[i], lit.m_c[i]);
    }
    return 0;
}
```

3、枚举类型-->enum

3.1 枚举类型的相关介绍

```
1. 枚举类型是一个基本的类型，关键字为enum.
    枚举类型的每个成员都是一个常量，枚举类型中的成员可以直接使用，
    枚举类型中的成员当成一个常量使用。

2. 定义枚举类型
    typedef enum  /*枚举类型名*/ {
        成员名0,
        成员名1,
        成员名2,
        ....
        成员名n,
    } 枚举类型别名;

    可以使用“枚举类型别名”或者“enum  枚举类型名”定义枚举类型的变量。

3. 枚举类型中的成员的使用：
    枚举类型中的成员可以直接使用，当成一个整型常量使用。

4. 定义枚举类型的变量，并初始化
    枚举类型别名  变量名 = 成员名;
                // 使用枚举类型中的成员对枚举类型的变量初始化


    enum  枚举类型名 变量名 = 整型常量;
                // 使用常量对枚举类型的变量进行初始化。

    枚举类型的变量名表示初始化常量值。

5. 枚举类型在开发中经常使用，可以作为函数的参数使用，或者函数的返回值。
```

```c
#include <stdio.h>
// 定义枚举类型
typedef enum {
    // 枚举中的成员默认第一个成员对应的常量值为0，
    // 后边的成员在前一个成员对应的常量值的基础之上加1
    ZERO,       // 0
    ONE,        // 1
    TWO,        // 2
    THREE,      // 3
    FOUR,       // 4
    FIVE,       // 5
}Number_t;

int main(int argc, const char *argv[])
{
    /*your code*/
    // 2. 枚举类型的中的成员可以直接使用，当成常量使用即可
    printf("ONE = %d\n", ONE);

```

```
    // 3. 定义枚举类型的变量，使用枚举类型中的成员进行初始化
    Number_t  num = FIVE;       // 常用这种方式对枚举类型的变量进行初始化
    printf("num = %d\n", num);

    // 4. 定义枚举类型的变量，使用整型常量进行赋值操作
    Number_t num1 = 8;       // 一般不这样使用
    printf("num1 = %d\n", num1);

    return 0;
}
```

```
#include <stdio.h>
// 定义枚举类型
typedef enum {
    // 枚举中的成员默认第一个成员对应的常量值为0，
    // 后边的成员在前一个成员对应的常量值的基础之上加1
    ZERO,       // 0
    ONE,        // 1
    TWO,        // 2
    THREE,      // 3
    FOUR,       // 4
    FIVE,       // 5
    // 定义枚举类型时，可以对部分成员进行初始化

    SENEN = 7,  // 7
    EIGHT,      // 8
    TEN  = 10,  // 10

}Number_t;

int main(int argc, const char *argv[])
{
    /*your code*/
    // 2. 枚举类型的中的成员可以直接使用，当成常量使用即可
    printf("ONE = %d\n", ONE);

    // 3. 定义枚举类型的变量，使用枚举类型中的成员进行初始化
    Number_t  num = FIVE;       // 常用这种方式对枚举类型的变量进行初始化
    printf("num = %d\n", num);

    // 4. 定义枚举类型的变量，使用整型常量进行赋值操作
    Number_t num1 = 8;       // 一般不这样使用
    printf("num1 = %d\n", num1);

    printf("EIGHT = %d\n", EIGHT);
    printf("TEN = %d\n", TEN);

    return 0;
}
```

```
#include <stdio.h>

```

```c
// 1. 定义枚举类型
typedef enum {
    macOS,
    Linux,
    Windows,
    Android,
    IOS,
}OS_t;
// 函数的参数和返回值都是一个枚举类型
// 参数决定打开那个操作系统，调用函数时，直接传递枚举类型的成员
OS_t  open_system(OS_t os)
{
    switch (os)
    {
    case macOS:
        printf("打开macOS系统\n");
        break;
    case Linux:
        printf("打开Linux系统\n");
        break;
    case Windows:
        printf("打开Windows系统\n");
        break;
    case Android:
        printf("打开Android系统\n");
        break;
    case IOS:
        printf("打开IOS系统\n");
        break;
    default:
        printf("不存在的操作系统\n");
    }
    return os;
}

int main(int argc, const char *argv[])
{
    /*your code*/
    // 调用函数，传递枚举类型的成员
    OS_t os = open_system(Linux);
    // 枚举类型的成员可以直接使用。
    if ( os == Linux)
    {
        printf("linux系统打开成功\n");
    }
    else if (os == macOS)
    {
        printf("masOS系统打开成功\n");
    }
    else if (os == Windows)
    {
        printf("windows系统打开成功\n");
    }
    else if (os == Android)
    {
```

```
        printf("Android系统打开成功\n");
    }
    else if (os == IOS)
    {
        printf("IOS系统打开成功\n");
    }
    else
    {
        printf("系统打开失败\n");
    }

    return 0;
}
```

Makefile

1、什么是Makefile

```
1. Makefile就是一个用来进行工程管理的文本文件，
    文本文件的名字叫做Makefile.
    Makefile文件中主要存放的是关于工程的配置和编译相关的命令。

2. Makefile文件的名字首字母一般大写，
    也写写成小写的makefile.

3. 如果在一个工程目录中，同时存在大写的Makefile和小写的Makefile
    当执行make命令时，默认解析的是小写的makefile文件。
```

2、什么是make

```
make是一个shell命令，专门用来解析Makefile文件，
当在终端执行make命令时，默认会解析Makefile文件中的规则，
最终执行对应的命令，完成对工程的配置和编译。
```

3、学习Makefile的要求

```
1. 必须掌握程序的构造的过程：预处理-》编译-》汇编-》连接

2. 需要具备面向依赖的思想。
    a.out <-依赖- ***.o <-依赖- ***.s <-依赖- ***.i <-依赖- ***.c

    只有这个依赖关系成立，最终才可以生成对应的可执行文件，
    如果依赖关系不成立，则不可以生成对应的可执行文件。

3. 具备多文件编程的思想
```

4、Makefile文件执行的过程

```
1. 如果没有Makefile文件，如何编译程序
```

```
    gcc 1.c  2.c  -o a.out

    优点：容易理解
    缺点：效率低，只要有一个.c文件被修改所有的.c文件都将被重新编译，
        最终生成对应的可执行文件.

2. 使用Makefile管理工程(采用分布的思想)
    2.1> -c : 只编译不链接，将每个.c文件生成对应的.o文件
    gcc -c 1.c -o 1.o
    gcc -c 2.c -o 2.o

    2.2> 将多个.o文件链接生成对应的可执行文件
    gcc 1.o 2.o -o a.out

    缺点：麻烦，需要学习Makefile文件的编写规则
    优点：执行效率高，
        Makefile根据文件的时间戳决定文件是否被编译。
        只要对应的源文件没有被修改则不会重新编译生成对应的.o文件，
        只会根据文件的时间戳编译被修改的文件，
        最后再将所有的.o文件链接生成一个可执行文件。
```

5、第一个Makefile文件

```
1. 创建一个Makefile文件，每个项目都需要单独的一个工程目录进行管理
    mkdir O1Project
    cd O1project
    touch hello.c           ---> 自己填充代码
    touch Makefile

2. 打开Makefile文件，添加以下内容：
hello<-- hello.o <-- hello.s <-- hello.i <-- hello.c
       |          |           |           |---> gcc -E hello.c -o hello.i
       |          |           |---> gcc -S hello.i -o hello.s
       |          |---> gcc -c hello.s -o hello.o
       |----> gcc hello.o -o hello

# Makefile文件中的规则
# 目标:依赖
#   shell命令

hello:hello.o
    gcc hello.o -o hello
hello.o:hello.s
    gcc -c hello.s -o hello.o

hello.s:hello.i
    gcc -S hello.i -o hello.s

# 如果依赖关系成立，则值对应规则下标的命令，
# 如果依赖关系不成立，则继续解析其他的依赖关系，
# 直到依赖关系成立。
hello.i:hello.c
    @# 在规则中，每个shell命令前边必须是一个tab键，
```

```
    @# 不可以使用4个空格替换TAB键
    gcc -E hello.c -o hello.i

# 规则可知没有依赖，此种规则只是为了完成某种操作
clean:
    rm *.[^c]  hello

3. 通过make命令执行Makefile文件
    make  目标名    ---> 执行此目标名对应的规则
            举例：make hello        ---> 最终生成hello可知程序
                make hello.s    ---> 最终生成hello.s可知程序

    make        ---> 省略目标
            默认执行的是Makefile文件中的第一个规则对应的目标
```

```
作业：为学生成绩管理系统编译一个Makefile文件。
```