

1、书接上回 --> 结构体

- 1.1 定义结构体指针数组
- 1.2 定义结构体数组指针
- 1.3 结构体中的成员包含指针类型的成员
- 1.4 结构体中的成员包含函数指针类型的成员
- 1.5 typedef定义新的结构体类型
- 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
 - 1.6.2 64位操作系统的结构体内存对齐

2、联合体/共用体 ---> union

- 2.1 联合体相关的介绍
- 2.2 联合体的练习题

3、枚举类型-->enum

- 3.1 枚举类型的相关介绍

Makefile

1、什么是Makefile

2、什么是make

3、学习Makefile的要求

4、Makefile文件执行的过程

5、第一个Makefile文件

1、书接上回 --> 结构体

1.1 定义结构体指针数组

```
1  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
3  2. 定义结构体指针数组
4      struct 结构体名 * 结构体指针数组名[元素个数];
5
6  3. 结构体指针数组的初始化
7      struct 结构体名 * 结构体指针数组名[元素个数] =
8          {&普通结构体变量0, &普通结构体变量1, .... }
9
10 4. 结构体指针数组中每个成员的访问
11     结构体指针数组名[下标]->成员变量名;
```

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5  // 1. 声明一个结构体类型
6  struct Teacher {
7      char name[20];
8      int age;
9      float salary;
10 };
11
```

```

12 void print_info(struct Teacher *p[], int len)
13 // void print_info(struct Teacher **p, int len)
14 {
15     for (int i = 0; i < len; i++)
16     {
17         printf("姓名: %s\t", p[i]->name);
18         printf("年龄: %d\t", p[i]->age);
19         printf("薪资: %f\n", p[i]->salary);
20     }
21     for (int i = 0; i < len; i++)
22     {
23         printf("姓名: %s\t", (*p[i]).name);
24         printf("年龄: %d\t", (*p[i]).age);
25         printf("薪资: %f\n", (*p[i]).salary);
26     }
27 }
28 int main(int argc, const char *argv[])
29 {
30     /*your code*/
31     // 定义结构体变量
32     struct Teacher tea1 = {"xiaozhou", 18, 20000},
33     tea2 = {
34         .name = "xiaodai",
35         .age = 18,
36         .salary = 25000
37     };
38
39     // 2. 定义一个结构体指针数组
40     struct Teacher * p_array[2] = {&tea1, &tea2};
41
42     print_info(p_array, 2);
43     return 0;
44 }

```

1.2 定义结构体数组指针

- 1 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
- 3 2. 定义结构体数组指针
- 4 struct 结构体名 (*结构体数组指针变量名)[列宽];
- 5
- 6 3. 结构体数组指针的初始化
- 7 struct 结构体名 二维数组名[行宽][列宽] = {};
- 8 struct 结构体名 (*结构体数组指针变量名)[列宽] = 二维数组名;
- 9
- 10 4. 结构体数组指针指向的二维数组的成员的访问
- 11 4.1> 将结构体数组指针当成二维数组名使用
- 12 结构体数组指针变量名[行下标][列下标].成员变量名;
- 13
- 14 4.2> 采用地址加偏移量的方式
- 15 (* (结构体数组指针变量名 + 行偏移) + 列偏移)->成员变量名;
- 16
- 17 ((* (结构体数组指针变量名 + 行偏移) + 列偏移)).成员变量名;

```

1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5  // 1. 声明一个结构体类型
6  struct Teacher {
7      char name[20];
8      int age;
9      float salary;
10 };
11
12 void print_info(struct Teacher (*arr_p)[2], int row, int col)
13 {
14     #if 0
15         for (int i = 0; i < row ; i++)
16         {
17             for (int j = 0; j < col ; j++)
18             {
19                 // 1. 将数组指针变量名当成二维数组名使用即可
20                 printf("姓名: %s\t", arr_p[i][j].name);
21                 printf("年龄: %d\t", arr_p[i][j].age);
22                 printf("薪资: %f\n", arr_p[i][j].salary);
23             }
24         }
25     #endif
26     #if 0
27         for (int i = 0; i < row ; i++)
28         {
29             for (int j = 0; j < col ; j++)
30             {
31                 // 1. 利用地址偏移的方式
32                 printf("姓名: %s\t", (*(arr_p + i) + j).name);
33                 printf("年龄: %d\t", (*(arr_p + i) + j).age);
34                 printf("薪资: %f\n", (*(arr_p + i) + j).salary);
35             }
36         }
37     #endif
38     for (int i = 0; i < row ; i++)
39     {
40         for (int j = 0; j < col ; j++)
41         {
42             // 1. 利用地址偏移的方式
43             printf("姓名: %s\t", (*(arr_p + i) + j)->name);
44             printf("年龄: %d\t", (*(arr_p + i) + j)->age);
45             printf("薪资: %f\n", (*(arr_p + i) + j)->salary);
46         }
47     }
48 }
49 int main(int argc, const char *argv[])
50 {
51     /*your code*/
52     // 2. 定义一个结构体一维数组
53     struct Teacher tea_arr[2] = {
54         [0] = {

```

```

55         .name = "xiaozhang",
56         .age = 20,
57         .salary = 15000,
58     },
59     [1] = {
60         .name = "xiaoli",
61         .age = 22,
62         .salary = 16000,
63     },
64 };
65
66 // 3. 定义结构体数组指针,指向2列的二维数组
67 struct Teacher (*array_p)[2] = &tea_arr;
68 print_info(array_p, 1, 2);
69
70 // 4. 定义一个2行2列的二维数组
71 struct Teacher tea_arr2[2][2] =
72 {
73     [0][0] = {
74         .name = "xiaoli",
75         .age = 22,
76         .salary = 16000,
77     },
78     [0][1] = {
79         .name = "xiaosun",
80         .age = 23,
81         .salary = 17000,
82     },
83     [1][0] = {
84         .name = "xiaowu",
85         .age = 24,
86         .salary = 17000,
87     },
88     [1][1] = {
89         .name = "xiaoqiao",
90         .age = 25,
91         .salary = 18000,
92     },
93 };
94 array_p = tea_arr2;
95 print_info(array_p, 2, 2);
96 return 0;
97 }

```

1.3 结构体中的成员包含指针类型的成员

```

1 struct 结构体名 {
2     数据类型 *指针变量名;    // 指向普通的变量, 指向堆区空间
3 };

```

```

1 #include <stdio.h>
2 #include <string.h>

```

```

3 #include <stdlib.h>
4
5 // 1. 声明结构体类型
6 struct Person{
7     char *name;
8     char sex;
9     int age;
10 };
11
12 void print_info(struct Person *per)
13 {
14     printf("姓名: %s\t性别: %c\t年龄: %d\n",
15           per->name, per->sex, per->age);
16 }
17 int main(int argc, const char *argv[])
18 {
19     /*your code*/
20     // 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。
21     char name[20] = "xiaozhao";
22     struct Person per1 = {name, 'M', 18};
23     print_info(&per1);
24
25     // 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
26     struct Person per2;
27     per2.name = (char *)malloc(sizeof(char)*20);
28     strcpy(per2.name, "xiaoqian");
29     per2.age = 19;
30     per2.sex = 'W';
31     print_info(&per2);
32
33     free(per2.name);
34     per2.name = NULL;
35
36     // 4. 定义结构体指针变量，在堆区分配空间，
37     // 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。
38
39     struct Person *per_p =
40         (struct Person *)malloc(sizeof(struct Person));
41     per_p->name = (char *)malloc(sizeof(char)*20);
42
43     strcpy(per_p->name, "xiaoli");
44     per_p->sex = 'M';
45     per_p->age = 20;
46     print_info(per_p);
47
48     free(per_p->name);
49     per_p->name = NULL;
50     free(per_p);
51     per_p = NULL;
52
53     return 0;
54 }

```

1.4 结构体中的成员包含函数指针类型的成员

```
1 struct 结构体名{
2     返回类型 (*函数指针变量名)(参数列表);
3 };
```

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4
5 struct Complex {
6     int m_i;
7     int m_j;
8 };
9
10 struct Cal{
11     struct Complex (*func_p)(struct Complex l, struct Complex r);
12 };
13
14 struct Complex add_func(struct Complex l, struct Complex r)
15 {
16     struct Complex sum;
17     sum.m_i = l.m_i + r.m_i;
18     sum.m_j = l.m_j + r.m_j;
19     return sum;
20 }
21 int main(int argc, const char *argv[])
22 {
23     /*your code*/
24     // 定义结构体变量
25     struct Complex c1 = {10,20}, c2 = {30, 40}, sum;
26     struct Cal cal;
27     // 对结构体中的成员函数指针变量进行初始化
28     cal.func_p = add_func;
29
30     // 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。
31     sum = cal.func_p(c1, c2);
32     printf("%d + %di\n", sum.m_i, sum.m_j);
33     return 0;
34 }
```

1.5 typedef定义新的结构体类型

```
1 typedef struct /*结构体名 (可以省略不写)*/ {
2     结构体成员;
3 } 结构体类型名;
4
5 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，
6 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。
7
```

```

1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4  // 使用typedef对结构类型起别名，重新定义新的结构体类型
5  typedef struct /*Complex*/ {
6      int m_i;
7      int m_j;
8  }Complex_t;
9
10 typedef struct /*cal*/ {
11     Complex_t (*func_p)(Complex_t l, Complex_t r);
12 }Cal_t;
13
14 Complex_t add_func(Complex_t l, Complex_t r)
15 {
16     Complex_t sum;
17     sum.m_i = l.m_i + r.m_i;
18     sum.m_j = l.m_j + r.m_j;
19     return sum;
20 }
21 int main(int argc, const char *argv[])
22 {
23     /*your code*/
24     // 定义结构体变量
25     Complex_t c1 = {10,20}, c2 = {30, 40}, sum;
26     Cal_t cal;
27     // 对结构体中的成员函数指针变量进行初始化
28     cal.func_p = add_func;
29
30     // 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。
31     sum = cal.func_p(c1, c2);
32     printf("%d + %d\n", sum.m_i, sum.m_j);
33     return 0;
34 }

```

```

1  // 声明结构体类型
2  typedef struct Person {
3      char *name;
4      char sex;
5      int age;
6  }Person_t;
7
8  // 结构体中嵌套一个结构体类型的指针变量。
9  typedef struct Student {
10     struct Person *per;
11     float score;
12 }Student_t;
13
14 使用“Student_t”定义结构体指针变量，在堆区分配空间，
15 然后对per成员在堆区分配空间，并初始化，
16 在对name成员在堆区分配空间，并初始化。

```

```

1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5  // 声明结构体类型
6  typedef struct Person {
7      char *name;
8      char sex;
9      int age;
10 }Person_t;
11
12 // 结构体中嵌套一个结构体类型的指针变量。
13 typedef struct Student {
14     struct Person *per;
15     float score;
16 }Student_t;
17
18 int main(int argc, const char *argv[])
19 {
20     /*your code*/
21     // 1. 定义结构体指针类型的变量，并在堆区分配空间
22     Student_t *stu_p = NULL;
23     stu_p = (Student_t *)malloc(sizeof(Student_t));
24
25     // 对stu_p->per成员分配堆区空间
26     stu_p->per = (Person_t *) malloc(sizeof(Person_t));
27
28     // 对stu_p->per->name成员分配堆区空间
29     stu_p->per->name = (char *)malloc(sizeof(char)*20);
30
31     // 2. 对结构体指针变量指向的堆区空间进行初始化
32     strcpy(stu_p->per->name, "zhangsan");
33     stu_p->per->sex = 'M';
34     stu_p->per->age = 18;
35     stu_p->score = 99;
36
37     // 3. 打印结果
38     printf("%s %c %d %f\n", stu_p->per->name,
39             stu_p->per->sex, stu_p->per->age,
40             stu_p->score);
41
42     // 4. 释放堆区的空间
43     free(stu_p->per->name);
44     stu_p->per->name = NULL;
45     free(stu_p->per);
46     stu_p->per = NULL;
47     free(stu_p);
48     stu_p = NULL;
49     return 0;
50 }

```

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

```
1 #include <stdio.h>
2
3 typedef struct {
4     char a[5];
5     char b;
6     char c;
7 }A_t;
8
9 typedef struct {
10     char a[5];           // 5字节
11                          // 保留1字节
12     short b;             // 2字节
13     char c;              // 1字节
14                          // 保留1字节
15 }B_t;
16
17 typedef struct {
18     char a[5];           // 5字节
19                          // 保留1字节
20     short b;             // 2字节
21     char c;              // 1字节
22                          // 保留3字节
23     int d;               // 4字节
24     int *p;              // 4字节
25 }C_t;
26
27 typedef struct {
28     char a[5];           // 5字节
29                          // 保留3字节
```

```

30     int b;           // 4字节
31     char c;          // 1字节
32                     // 保留3字节
33     long long int d; // 8字节
34 }D_t;
35 typedef struct {
36     char a[5];        // 5字节
37                     // 保留3字节
38     int b;            // 4字节
39     long long int c; // 8字节
40     char d;           // 1字节
41                     // 保留3字节
42 }E_t;
43
44 int main(int argc, const char *argv[])
45 {
46     /*your code*/
47     // 1> 如果结构体中的成员最大的成员占1个字节的空间，
48     // 则结构体的大小为1的整数倍
49     // 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
50     // char / unsigned char
51
52     printf("A_t size = %d\n", sizeof(A_t));
53
54     // 2> 如果结构体中的成员最大的成员占2个字节的空间，
55     // 则结构体的大小为2的整数倍
56     // 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
57     // short / unsigned short
58     printf("B_t size = %d\n", sizeof(B_t));
59     B_t B;
60     printf("&B_t.a address = %p\n", &B.a);
61     printf("&B_t.b address = %p\n", &B.b);
62     printf("&B_t.c address = %p\n", &B.c);
63
64     // 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，
65     // 则结构体的大小为4的整数倍
66     // 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
67     // int / unsigned int / long int / float / 指针 / 函数指针
68     // double / long long int
69     printf("C_t size = %d\n", sizeof(C_t));
70     C_t C;
71     printf("&C_t.a address = %p\n", &C.a);
72     printf("&C_t.b address = %p\n", &C.b);
73     printf("&C_t.c address = %p\n", &C.c);
74     printf("&C_t.d address = %p\n", &C.d);
75     printf("&C_t.p address = %p\n", &C.p);
76
77     printf("D_t size = %d\n", sizeof(D_t));
78     D_t D;
79     printf("&D_t.a address = %p\n", &D.a);
80     printf("&D_t.b address = %p\n", &D.b);
81     printf("&D_t.c address = %p\n", &D.c);
82     printf("&D_t.d address = %p\n", &D.d);
83
84     printf("E_t size = %d\n", sizeof(E_t));

```

```

85     E_t  E;
86     printf("&E_t.a address = %p\n", &E.a);
87     printf("&E_t.b address = %p\n", &E.b);
88     printf("&E_t.c address = %p\n", &E.c);
89     printf("&E_t.d address = %p\n", &E.d);
90     return 0;
91 }

```

1.6.2 64位操作系统的结构体内存对齐

- 1 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，
2 构造类型也是由基本类型构成。
- 3
- 4 2. 声明结构体类型的大小
 - 5 1> 如果结构体中的成员最大的成员占1个字节的空间，
6 则结构体的大小为1的整数倍
 - 7 2> 如果结构体中的成员最大的成员占2个字节的空间，
8 则结构体的大小为2的整数倍
 - 9 3> 如果结构体中的成员最大的成员占4个字节的空间，
10 则结构体的大小为4的整数倍
 - 11 4> 如果结构体中的成员最大的成员占8个字节的空间，
12 则结构体的大小为8的整数倍
- 13
- 14 3. 结构体中成员的地址
 - 15 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
16 char / unsigned char
 - 17 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
18 short / unsigned short
 - 19 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
20 int / unsigned int / float
 - 21 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
22 double / long long int / long int / 指针 / 函数指针
 - 23

```

1  #include <stdio.h>
2
3  typedef struct {
4      char a[5];
5      char b;
6      char c;
7  }A_t;
8
9  typedef struct {
10     char a[5];           // 5字节
11                           // 保留1字节
12     short b;             // 2字节
13     char c;              // 1字节
14                           // 保留1字节
15 }B_t;
16
17 typedef struct {
18     char a[5];           // 5字节

```

```

19         // 保留1字节
20     short b;    // 2字节
21     char c;     // 1字节
22         // 保留3字节
23     int d;      // 4字节
24     int *p;     // 8字节
25 }C_t;
26
27 typedef struct {
28     char a[5];   // 5字节
29         // 保留3字节
30     int b;       // 4字节
31     char c;      // 1字节
32         // 保留3字节
33     long long int d; // 8字节
34 }D_t;
35 typedef struct {
36     char a[5];   // 5字节
37         // 保留3字节
38     int b;       // 4字节
39         // 保留4字节
40     long long int c; // 8字节
41     char d;      // 1字节
42         // 保留7字节
43 }E_t;
44
45 int main(int argc, const char *argv[])
46 {
47     /*your code*/
48     // 1> 如果结构体中的成员最大的成员占1个字节的空间，
49     // 则结构体的大小为1的整数倍
50     // 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
51     // char / unsigned char
52
53     printf("A_t size = %ld\n", sizeof(A_t));
54
55     // 2> 如果结构体中的成员最大的成员占2个字节的空间，
56     // 则结构体的大小为2的整数倍
57     // 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
58     // short / unsigned short
59     printf("B_t size = %ld\n", sizeof(B_t));
60     B_t B;
61     printf("&B_t.a address = %p\n", &B.a);
62     printf("&B_t.b address = %p\n", &B.b);
63     printf("&B_t.c address = %p\n", &B.c);
64
65     // 3> 如果结构体中的成员最大的成员占4个字节的空间，
66     // 则结构体的大小为4的整数倍
67     // 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
68     // int / unsigned int / float
69     printf("C_t size = %ld\n", sizeof(C_t));
70     C_t C;
71     printf("&C_t.a address = %p\n", &C.a);
72     printf("&C_t.b address = %p\n", &C.b);
73     printf("&C_t.c address = %p\n", &C.c);

```

```

74     printf("&C_t.d address = %p\n", &C.d);
75     printf("&C_t.p address = %p\n", &C.p);
76
77     // 4> 如果结构体中的成员最大的成员占8个字节的空间,
78     // 则结构体的大小为8的整数倍
79     // 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
80     // double / long long int / long int / 指针 / 函数指针
81     printf("D_t size = %ld\n", sizeof(D_t));
82     D_t D;
83     printf("&D_t.a address = %p\n", &D.a);
84     printf("&D_t.b address = %p\n", &D.b);
85     printf("&D_t.c address = %p\n", &D.c);
86     printf("&D_t.d address = %p\n", &D.d);
87
88     printf("E_t size = %ld\n", sizeof(E_t));
89     E_t E;
90     printf("&E_t.a address = %p\n", &E.a);
91     printf("&E_t.b address = %p\n", &E.b);
92     printf("&E_t.c address = %p\n", &E.c);
93     printf("&E_t.d address = %p\n", &E.d);
94     return 0;
95 }

```

2、联合体/共用体 ---> union

2.1 联合体相关的介绍

```

1  1. 联合体概念
2      联合体是一个构造类型, 关键字union.
3      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。
4
5      联合体类型的大小为联合体中最大类型成员的整数倍,
6      联合体也需要考虑内存对齐的问题。
7
8  2. 声明联合体类型
9      typedef union /*联合体名(可以省略不写)*/ {
10         数据类型 成员变量名1;
11         数据类型 成员变量名2;
12         .....
13         数据类型 成员变量名n;
14     } 联合体类型别名;
15
16     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。
17         联合体类型别名 变量名;
18         union 联合体名 *指针变量名;
19
20  3. 定义联合体类型的变量, 并进行初始化
21         联合体类型别名 变量名;
22         union 联合体名 *指针变量名 = &变量名;
23
24         变量名.成员名 = 初始值;
25         指针变量名->成员名 = 初始值;
26
27  4. 访问联合体类型变量中的成员

```

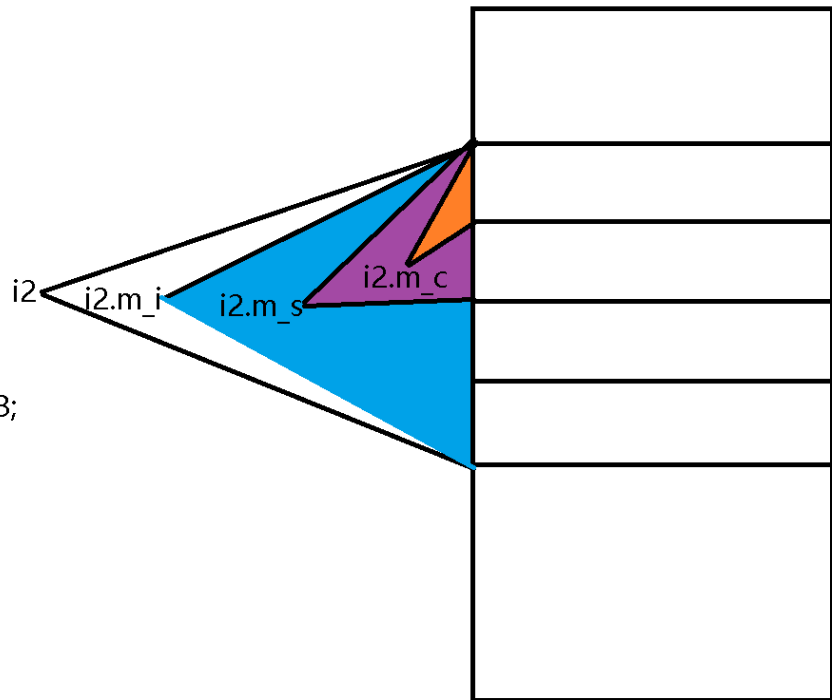
```
28     变量名.成员名;
29     指针变量名->成员名;
30
31 5. 在目前开发中,基本上不在使用联合体,联合体中的成员共用一块内存空间,
32     相对来说如果使用不当,就是不安全。
33     但是在内核的底层代码中依然存在着部分联合体的使用。
```

2.2 联合体的练习题

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5  // 1. 声明联合体类型
6  typedef union {
7      int m_i;
8      short m_s;
9      char m_c;
10 } Integer_t;          // 4字节
11
12 typedef union {
13     int m_i;
14     char c[5];
15 } data_t;             // 8字节
16
17 int main(int argc, const char *argv[])
18 {
19     /*your code*/
20     // 1. 验证联合体类型的大小为最大成员的整数倍
21     printf("Integer_t size = %ld\n", sizeof(Integer_t));
22     printf("data_t size = %ld\n", sizeof(data_t));
23
24     // 2. 联合体类型中的成员公用同一块内存空间
25     Integer_t i1;
26     printf("i1.m_i address = %p\n", &i1.m_i);
27     printf("i1.m_s address = %p\n", &i1.m_s);
28     printf("i1.m_c address = %p\n", &i1.m_c);
29
30     // 3. 定义联合体类型的变量并对成员进行初始化,
31     // 一般使用联合体的哪个成员就对那个成员进行初始化。
32     Integer_t i2;
33     i2.m_i = 0x12345678;
34     printf("i2.m_i = %#x\n", i2.m_i);    // 0x12345678
35     printf("i2.m_s = %#x\n", i2.m_s);    // 0x5678
36     printf("i2.m_c = %#x\n", i2.m_c);    // 0x78
37
38     // 4. 定义联合类型的指针变量
39     Integer_t * i_p = (Integer_t *)malloc(sizeof(Integer_t));
40     i_p->m_i = 0x12345678;
41     printf("i_p->m_i = %#x\n", i_p->m_i);    // 0x12345678
42     printf("i_p->m_s = %#x\n", i_p->m_s);    // 0x5678
43     printf("i_p->m_c = %#x\n", i_p->m_c);    // 0x78
44     return 0;
```

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

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



```
1  练习题:
2      声明一个联合体类型，测试计算机的大小端存储的问题。
3
4  typedef union little {
5      unsigned int m_i;
6      unsigned char m_c[4];
7  }little_t;
8
9      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。
10
11 #include <stdio.h>
12 // 声明一个联合体类型
13 typedef union little {
14     unsigned int m_i;
15     unsigned char m_c[4];
16 }little_t;
17
18 int main(int argc, const char *argv[])
19 {
20     /*your code*/
21     // 定义联合体类型，并进行初始化
22     little_t lit;
23     lit.m_i = 0x12345678;
24     // 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
25     // 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
26     for (int i = 0; i < 4; i++)
27     {
28         printf("&lit.m_c[%d] = %p = %#x\n", i, &lit.m_c[i], lit.m_c[i]);
29     }
30     return 0;
31 }
```

3、枚举类型-->enum

3.1 枚举类型的相关介绍

```
1  1. 枚举类型是一个基本的类型，关键字为enum。
2      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，
3      枚举类型中的成员当成一个常量使用。
4
5  2. 定义枚举类型
6      typedef enum  /*枚举类型名*/ {
7          成员名0,
8          成员名1,
9          成员名2,
10         ....
11         成员名n,
12     } 枚举类型别名;
13
14     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。
15
16 3. 枚举类型中的成员的使用：
17     枚举类型中的成员可以直接使用，当成一个整型常量使用。
18
19 4. 定义枚举类型的变量，并初始化
20     枚举类型别名 变量名 = 成员名;
21         // 使用枚举类型中的成员对枚举类型的变量初始化
22
23
24     enum 枚举类型名 变量名 = 整型常量;
25         // 使用常量对枚举类型的变量进行初始化。
26
27     枚举类型的变量名表示初始化常量值。
28
29 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。
```

```
1  #include <stdio.h>
2  // 定义枚举类型
3  typedef enum {
4      // 枚举中的成员默认第一个成员对应的常量值为0，
5      // 后边的成员在前一个成员对应的常量值的基础之上加1
6      ZERO,          // 0
7      ONE,            // 1
8      TWO,            // 2
9      THREE,          // 3
10     FOUR,           // 4
11     FIVE,            // 5
12 }Number_t;
13
14 int main(int argc, const char *argv[])
15 {
16     /*your code*/
17     // 2. 枚举类型的中的成员可以直接使用，当成常量使用即可
18     printf("ONE = %d\n", ONE);
19 }
```



```

20 // 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
21 Number_t num = FIVE; // 常用这种方式对枚举类型的变量进行初始化
22 printf("num = %d\n", num);
23
24 // 4. 定义枚举类型的变量，使用整型常量进行赋值操作
25 Number_t num1 = 8; // 一般不这样使用
26 printf("num1 = %d\n", num1);
27
28 return 0;
29 }

```

```

1 #include <stdio.h>
2 // 定义枚举类型
3 typedef enum {
4     // 枚举中的成员默认第一个成员对应的常量为0，
5     // 后边的成员在前一个成员对应的常量的基础之上加1
6     ZERO, // 0
7     ONE, // 1
8     TWO, // 2
9     THREE, // 3
10    FOUR, // 4
11    FIVE, // 5
12    // 定义枚举类型时，可以对部分成员进行初始化
13
14    SEVEN = 7, // 7
15    EIGHT, // 8
16    TEN = 10, // 10
17
18 }Number_t;
19
20 int main(int argc, const char *argv[])
21 {
22     /*your code*/
23     // 2. 枚举类型的中的成员可以直接使用，当成常量使用即可
24     printf("ONE = %d\n", ONE);
25
26     // 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
27     Number_t num = FIVE; // 常用这种方式对枚举类型的变量进行初始化
28     printf("num = %d\n", num);
29
30     // 4. 定义枚举类型的变量，使用整型常量进行赋值操作
31     Number_t num1 = 8; // 一般不这样使用
32     printf("num1 = %d\n", num1);
33
34     printf("EIGHT = %d\n", EIGHT);
35     printf("TEN = %d\n", TEN);
36
37     return 0;
38 }

```

```

1 #include <stdio.h>
2

```

```

3 // 1. 定义枚举类型
4 typedef enum {
5     macOS,
6     Linux,
7     windows,
8     Android,
9     IOS,
10 }OS_t;
11 // 函数的参数和返回值都是一个枚举类型
12 // 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
13 OS_t open_system(OS_t os)
14 {
15     switch (os)
16     {
17     case macOS:
18         printf("打开macOS系统\n");
19         break;
20     case Linux:
21         printf("打开Linux系统\n");
22         break;
23     case windows:
24         printf("打开windows系统\n");
25         break;
26     case Android:
27         printf("打开Android系统\n");
28         break;
29     case IOS:
30         printf("打开IOS系统\n");
31         break;
32     default:
33         printf("不存在的操作系统\n");
34     }
35     return os;
36 }
37
38 int main(int argc, const char *argv[])
39 {
40     /*your code*/
41     // 调用函数，传递枚举类型的成员
42     OS_t os = open_system(Linux);
43     // 枚举类型的成员可以直接使用。
44     if ( os == Linux)
45     {
46         printf("linux系统打开成功\n");
47     }
48     else if (os == macOS)
49     {
50         printf("masOS系统打开成功\n");
51     }
52     else if (os == windows)
53     {
54         printf("windows系统打开成功\n");
55     }
56     else if (os == Android)
57     {

```

```
58     printf("Android系统打开成功\n");
59 }
60 else if (os == IOS)
61 {
62     printf("IOS系统打开成功\n");
63 }
64 else
65 {
66     printf("系统打开失败\n");
67 }
68
69 return 0;
70 }
```

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

- 1 make是一个shell命令，专门用来解析Makefile文件，
2 当在终端执行make命令时，默认会解析Makefile文件中的规则，
3 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

```

2 gcc 1.c 2.c -o a.out
3
4 优点：容易理解
5 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，
6 最终生成对应的可执行文件。
7
8 2. 使用Makefile管理工程(采用分布的思想)
9 2.1> -c : 只编译不链接，将每个.c文件生成对应的.o文件
10 gcc -c 1.c -o 1.o
11 gcc -c 2.c -o 2.o
12
13 2.2> 将多个.o文件链接生成对应的可执行文件
14 gcc 1.o 2.o -o a.out
15
16 缺点：麻烦，需要学习Makefile文件的编写规则
17 优点：执行效率高，
18 Makefile根据文件的时间戳决定文件是否被编译。
19 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，
20 只会根据文件的时间戳编译被修改的文件，
21 最后再将所有的.o文件链接生成一个可执行文件。
22

```

5、第一个Makefile文件

```

1 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
2 mkdir 01Project
3 cd 01project
4 touch hello.c          ---> 自己填充代码
5 touch Makefile
6
7 2. 打开Makefile文件，添加以下内容：
8 hello<-- hello.o <-- hello.s <-- hello.i <-- hello.c
9 | | | |---> gcc -E hello.c -o hello.i
10 | | |---> gcc -S hello.i -o hello.s
11 | |---> gcc -c hello.s -o hello.o
12 |----> gcc hello.o -o hello
13
14 # Makefile文件中的规则
15 # 目标:依赖
16 # shell命令
17
18 hello:hello.o
19 gcc hello.o -o hello
20 hello.o:hello.s
21 gcc -c hello.s -o hello.o
22
23 hello.s:hello.i
24 gcc -S hello.i -o hello.s
25
26 # 如果依赖关系成立，则值对应规则下标的命令，
27 # 如果依赖关系不成立，则继续解析其他的依赖关系，
28 # 直到依赖关系成立。
29 hello.i:hello.c
30 @# 在规则中，每个shell命令前边必须是一个tab键，

```

```

31  @# 不可以使用4个空格替换TAB键
32  gcc -E hello.c -o hello.i
33
34  # 规则可知没有依赖，此种规则只是为了完成某种操作
35  clean:
36      rm *.[^\c] hello
37
38  3. 通过make命令执行Makefile文件
39      make 目标名  ---> 执行此目标名对应的规则
40          举例: make hello  ---> 最终生成hello可知程序
41              make hello.s  ---> 最终生成hello.s可知程序
42
43      make  ---> 省略目标
44          默认执行的是Makefile文件中的第一个规则对应的目标

```

1 | 作业：为学生成绩管理系统编译一个Makefile文件。

