

1、宏定义的使用

- 2.1 常量宏
- 2.2 宏函数的返回值
- 2.3 #ifdef...#else...#endif使用
- 2.4 #ifndef...#else...#endif使用
- 2.5 #if defined...#else...#endif使用
- 2.6 宏定义和do...while的结合
- 2.7 宏定义和#结合使用
- 2.7 宏定义和##结合使用

2、goto跳转语句的使用

- 2.1 语法格式
- 2.2 参考案例

3、在堆区分配内存空间

- 3.1 malloc和free函数的使用
- 3.2 参考案例

4、多文件的编程

- 4.1 多文件编程相关的概念
- 4.2 编写一个多文件编程的代码

5、6种存储类型

- 5.1 定义变量的格式
- 5.2 6种存储类型
- 5.3 auto存储类型 --> 自动类型
- 5.4 const存储类型 --> 常量
- 5.5 extern存储类型 --> 外部的
- 5.6 static存储类型 --> 静态
- 5.7 register存储类型 --> 寄存器
- 5.8 volatile存储类型 --> 易变的

6、作业

7、明天的授课内容

1、宏定义的使用

2.1 常量宏

```
1  #define PI 3.14
2  #define CH 'A'
3  #define STRING "hello world"
4
5  宏定义在预处理阶段进行替换。
6
7  如果在宏定义中使用算数运算时，尽量多加()，使用()将宏定义的值括起来；
8
9  // 圆的直径的宏定义
10 #define D(r) r + r
11
12 // 求圆的周长的宏定义
13 #define CIRCLE(r) D(r) * 3.14
14
15 // 宏定义的使用
16 CIRCLE(3) ----> 3 + 3 * 3.14          ---> 结果错误
```

```

17 -----
18 // 圆的直径的宏定义
19 #define D(r) ((r) + (r))
20
21 // 求圆的周长的宏定义
22 #define CIRCLE(r) D(r) * 3.14
23
24 // 宏定义的使用
25 CIRCLE(3) ----> ((3) + (3)) * 3.14

```

2.2 宏函数的返回值

- 1 1. 如果定义的宏函数，只有一条语句，宏定义的返回值直接使用即可。
2 `#define max_value(a, b) (((a) > (b)) ? (a) : (b))`
3
4 `int max = max_value(10, 20);`
5
6
- 7 2. 如果定义的宏函数中有多条语句，最后一条语句的结果被返回，
8 并且多条语句需要使用({语句1;语句2;语句2;...;})括起来。
9 宏函数的返回值不是通过return返回，而是最后一条语句的结果被返回。

```

1  #include <stdio.h>
2  // 使用宏定义的方式实现求两个数的最大值
3  #define MAX_VALUE(a,b) (a > b) ? (a) : (b)
4
5  // 带返回值的宏函数
6  #define MIN_VALUE(a,b) ({(a > b) ? (b) : (a);})
7
8  // 带返回值的宏函数，最后一个表达式的结果被返回，
9  // 不是用过return返回
10 #define MIN_VALUE1(a,b) ({int ret; \
11                             if (a > b) \
12                                 ret = b; \
13                             else \
14                                 ret = a; \
15                             ret;})
16
17
18 int main(int argc, const char *argv[])
19 {
20     /*your code*/
21     int max=MAX_VALUE(100,300);
22     printf("max value = %d\n",max);
23     int min=MIN_VALUE(100,300);
24     printf("min value = %d\n", min);
25
26     min=MIN_VALUE1(600,500);
27     printf("min value = %d\n", min);
28     return 0;
29 }

```

2.3 #ifdef...#else...#endif使用

```
1 C语言的注释手段:
2     //
3     /**/
4     #if 0/1
5     #else
6     #endif
7
8 #define 宏定义名
9
10 #ifdef 宏定义名
11     // 如果“宏定义名”被定义了, 则此段代码有效
12 #else
13     // 如果“宏定义名”没有被定义了, 则此段代码有效
14 #endif
```

```
1 #include <stdio.h>
2 #define DEBUG
3 int main(int argc, const char *argv[])
4 {
5     /* 操作系统提供的几个调试宏:
6      * __FILE__ : 表示文件名
7      * __func__ : 表示函数名
8      * __LINE__ : 表示行号
9      */
10    /*your code*/
11 #ifdef DEBUG
12     // 如果“宏定义名”被定义了, 则此段代码有效
13     printf("%s-%s-%d\n", __FILE__, __func__, __LINE__);
14 #else
15     // 如果“宏定义名”没有被定义了, 则此段代码有效
16     printf("%s-%s-%d\n", __FILE__, __func__, __LINE__);
17 #endif
18
19     return 0;
20 }
```

2.4 #ifndef...#else...#endif使用

```
1 #define 宏定义名
2
3 #ifndef 宏定义名
4     // 如果“宏定义名”没有被定义了, 则此段代码有效
5 #else
6     // 如果“宏定义名”被定义了, 则此段代码有效
7 #endif
```

```
1 #include <stdio.h>
2 #define DEBUG
3 int main(int argc, const char *argv[])
```

```

4 {
5     /* 操作系统提供的几个调试宏:
6      * __FILE__ : 表示文件名
7      * __func__ : 表示函数名
8      * __LINE__ : 表示行号
9      */
10    /*your code*/
11 #ifndef DEBUG
12     // 如果“宏定义名”没有被定义了, 则此段代码有效
13     printf("%s-%s-%d\n", __FILE__, __func__, __LINE__);
14 #else
15     // 如果“宏定义名”被定义了, 则此段代码有效
16     printf("%s-%s-%d\n", __FILE__, __func__, __LINE__);
17 #endif
18
19     return 0;
20 }

```

2.5 #if defined...#else...#endif使用

```

1 #if defined(宏定义名)
2     // 如果“宏定义名”被定义了, 则此段代码有效
3 #else
4     // 如果“宏定义名”没有被定义了, 则此段代码有效
5 #endif
6 -----
7 #if !defined(宏定义名)
8     // 如果“宏定义名”没有被定义了, 则此段代码有效
9 #else
10    // 如果“宏定义名”被定义了, 则此段代码有效
11 #endif
12 -----
13 #if defined(宏定义名1) && defined(宏定义名2)
14    // 如果“宏定义名1”和“宏定义名2”都被定义了, 则此段代码有效
15 #else
16    // 如果“宏定义名1”和“宏定义名2”只要有一个没有被定义, 则此段代码有效
17 #endif
18 -----
19 #if defined(宏定义名1) || defined(宏定义名2)
20    // 如果“宏定义名1”和“宏定义名2”只要有一个被定义了, 则此段代码有效
21 #else
22    // 如果“宏定义名1”和“宏定义名2”都没有被定义, 则此段代码有效
23 #endif
24
25 #if defined可以进行逻辑运算, 而#ifdef和#ifndef不可以进行逻辑运算。

```

```

1 #include <stdio.h>
2 #define DEBUG
3 int main(int argc, const char *argv[])
4 {
5     /* 操作系统提供的几个调试宏:
6      * __FILE__ : 表示文件名

```

```

7      * __func__ : 表示函数名
8      * __LINE__ : 表示行号
9      */
10     /*your code*/
11 #if defined(DEBUG)
12     // 如果“宏定义名”被定义了, 则此段代码有效
13     printf("%s-%s-%d\n", __FILE__, __func__, __LINE__);
14 #else
15     // 如果“宏定义名”没有被定义了, 则此段代码有效
16     printf("%s-%s-%d\n", __FILE__, __func__, __LINE__);
17 #endif
18
19 #if !defined(DEBUG)
20     // 如果“宏定义名”被定义了, 则此段代码有效
21     printf("%s-%s-%d\n", __FILE__, __func__, __LINE__);
22 #else
23     // 如果“宏定义名”没有被定义了, 则此段代码有效
24     printf("%s-%s-%d\n", __FILE__, __func__, __LINE__);
25 #endif
26
27     return 0;
28 }

```

2.6 宏定义和do...while的结合

1 | 进行宏定义时, 如果宏值中有多条C语言, 使用do{}while(0)包含一下。

```

1  案例: do...while循环和宏定义配合使用的案例。
2  #include <stdio.h>
3
4  // # : 将参数转换为字符串
5  // ## : 字符串的拼接
6
7  #define PRINT(str,err) printf("%s\n",#str);return err
8
9  // 宏定义默认要求写到1行, 如果分多行进行书写, 要求加续行符“\”
10 #define PRI_ERR(str,err) do { \
11     printf("%s\n", #str); \
12     return err; \
13 } while(0)
14
15 #define PRI_DEBUG(str,err) {printf("%s\n", #str); \
16     return err; \
17 }
18
19 int main(int argc, const char *argv[])
20 {
21     int retValue;
22     retValue = putchar('A');
23     if (retValue == -1)
24     {
25 #if 0
26         printf("put char failed\n");
27         return -1;

```

```

28 #endif
29     // 如果if分支只有1条语句可以省略花括号,
30     // 但是这里调用的宏定义, 展开之后有多条语句,
31     // 因此if的{}不可以省略。
32     PRINT(put char failed, -1);
33 }
34
35 retValue = putchar('B');
36 if (retValue == -1)
37     PRI_ERR(put char failed, -1);
38
39
40 retValue = putchar('C');
41 if (retValue == -1)
42     // PRI_DEBUG(send char failed, -1); // error
43     PRI_DEBUG(send char failed, -1) // OK
44 else
45     printf("send char success\n");
46
47 return 0;
48 }
49

```

2.7 宏定义和#结合使用

1 | 在宏定义中使用#时, 可以将传递的参数转换为字符串。

```

1 #include <stdio.h>
2 #define NAME1 "zhangsan"
3 #define NAME2(n) "n"
4 #define NAME3(n) n
5
6 // 在宏定义的值中的变量名前加#, 将变量的值转换为字符串
7 #define NAME4(n) #n
8 int main(int argc, const char *argv[])
9 {
10     /*your code*/
11     printf("my name is %s\n", "zhangsan");
12     printf("my name is %s\n", NAME1);
13     // ---> 预处理阶段进行替换为以下结果:
14     // printf("my name is %s\n", "zhangsan");
15
16     printf("my name is %s\n", NAME2(zhangsan));
17     // ---> 预处理阶段进行替换为以下结果:
18     // printf("my name is %s\n", "n");
19
20     printf("my name is %s\n", NAME3("zhangsan"));
21     // ---> 预处理阶段进行替换为以下结果:
22     // printf("my name is %s\n", "zhangsan");
23
24     printf("my name is %s\n", NAME4(zhangsan));
25     // ---> 预处理阶段进行替换为以下结果:
26     // printf("my name is %s\n", "zhangsan");
27     return 0;

```

2.7 宏定义和##结合使用

1 | 在宏定义中使用两个##,可以实现字符串的拼接。

```

1  #include <stdio.h>
2  // 实现pri和ntf字符串的拼接
3  #define DEBUG()  pri##ntf("test code\n")
4  // 将参数a,b对应的字符串进行拼接
5  #define SHOW(a,b)  a##b("test code\n")
6
7  int main(int argc, const char *argv[])
8  {
9      DEBUG();
10     // 替换结果  printf("test code\n");
11
12     SHOW(print, f);
13     // 替换结果  printf("test code\n");
14     return 0;
15 }
```

2、goto跳转语句的使用

2.1 语法格式

```

1  1. 语法格式
2      goto Label(标签);    ---> 跳转到Label标签下边的语句开始执行
3
4      Label:
5          C语句
6
7  2. 注意事项
8      1> goto语句只能在函数内进行跳转;
9      2> 在应用层开发中一般不使用goto跳转语句;
10     3> 在linux驱动开发中, 每个驱动中基本上都会看到goto的使用。
```

2.2 参考案例

```

1  案例1, 使用goto实现1-n之间的数据的求和?
2      及使用goto跳转语句实现一个循环的代码。
3  #include <stdio.h>
4
5  int main(int argc, const char *argv[])
6  {
7      /*your code*/
8      int sum = 0;
9      int n;
10     int i = 1;
```

```

11
12     printf("请输入要给整数给n变量 > ");
13     scanf("%d", &n);
14 loop:
15     sum = sum + i;
16     i++;
17     if (i <= n)
18     {
19         goto loop;
20     }
21
22     printf("sum = %d\n", sum);
23     return 0;
24 }

```

1 案例2: 定义一个unsigned int类型的整型变量num, 从终端对此变量进行初始化,

2 使用goto语句的方式实现将num变量转换为二进制进行输出。

3

4 利用位运算的方式: num & (0x1 << 31)

5

```

6 #include <stdio.h>
7
8 int main(int argc, const char *argv[])
9 {
10     /*your code*/
11     unsigned int num;
12     int i = 31;
13     printf("请输入一个整数 > ");
14     scanf("%u", &num);
15     printf("0b");
16 loop:
17     // if (num & (1 << i))
18     //     printf("1");
19     // else
20     //     printf("0");
21     printf("%d", num & (1 << i) ? 1 : 0);
22     i--;
23     if (i >= 0)
24         goto loop;
25
26     putchar('\n');
27     return 0;
28 }

```

3、在堆区分配内存空间

3.1 malloc和free函数的使用

```

1 man 3 malloc/free查看函数的帮助手册
2
3 #include <stdlib.h>
4
5 void *malloc(size_t size);

```



```
6  功能：手动在堆区分配内存空间
7  参数：
8      @ size : 分配堆区内存空间的大小
9              以字节为单位进行分配
10 返回值：
11      成功返回分配的堆区空间的首地址，
12      失败返回NULL
13
14 void free(void *ptr);
15 功能：手动释放堆区的空间
16 参数：
17      @ ptr : 释放的堆区空间的首地址
18 返回值：
19      无
20
21
```

3.2 参考案例

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4  int main(int argc, const char *argv[])
5  {
6      /*your code*/
7      // 回顾：定义指针变量如何进行初始化
8      #if 0
9          int *p = NULL;
10         int a = 100;    // 变量a在栈区分配空间
11         p = &a;    // 指针变量指向栈区的空间
12
13         char arr[10] = "hello"; // 字符数组，在栈区分配空间
14         char *str_p = arr;    // 字符指针指向栈区空间
15
16         char *str_p2 = "world"; // 字符指针指向字符串对应的常量区
17     #endif
18
19     /*
20     * 定义指针类型的变量，指向堆区的空间，
21     * 在堆区分配的空间，使用malloc手动在堆区分配空间，
22     * 使用堆区空间完成之后，需要使用free手动释放堆区空间，
23     * 即堆区的空间要求手动分配，手动释放。
24     * 如果堆区的空间没有手动的释放，当程序结束之后，
25     * 系统也会帮助我们回收堆区的空间。
26     */
27
28     // 1. 定义指针类型的变量，使用malloc在堆区分配空间
29     int *m_p = NULL;
30     m_p = (int *)malloc(sizeof(int));    // m_p指向堆区空间
31     if (m_p == NULL)
32     {
33         printf("malloc failed!\n");
34         printf("%s:%s:%d\n", __FILE__, __func__, __LINE__);
35         return -1;
36     }
37 }
```

```

36     }
37
38     // 2. 对m_p指向的堆区空间进行初始化
39     *m_p = 10086;    // 对堆区的空间赋值
40
41     printf("*m_p = %d\n", *m_p);
42
43     // 3. 使用free释放堆区的空间
44     free(m_p);
45
46     // 4. 释放完成之后一定要将指针变量赋值为NULL,
47     //      防止野指针的出现
48     m_p = NULL;
49
50     return 0;
51 }

```

```

1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4  int main(int argc, const char *argv[])
5  {
6      /*your code*/
7      // 1. 定义指针类型的变量, 指向堆区的空间
8      char *name = (char *)malloc(sizeof(char) * 20);
9      if (name == NULL)
10     {
11         printf("malloc failed!\n");
12         printf("%s:%s:%d\n", __FILE__, __func__, __LINE__);
13         return -1;
14     }
15     printf("堆区 name = %p\n", name);
16     // 2. 对name指向的堆区空间初始化
17     // name = "zhangsan"; // name指向字符串对应的常量区
18     // printf("常量区 name = %p\n", name);
19     strcpy(name, "zhangsan");
20     printf("%s\n", name);
21
22     // 3. 释放堆区的空间
23     printf("释放之前 name = %p\n", name);
24     // free释放堆区的空间, 释放完成堆区的空间之后,
25     // 并不会将name中的值清0(NULL)
26     free(name);
27     printf("释放之后 name = %p\n", name);
28
29     // 释放name指向的堆区空间之后, 如果name没有指向NULL,
30     // 依然可以通过name访问对应的堆区空间, 测试访问的就是
31     // 非法的内存的空间。
32     // 此时name就会变成一个野指针, 有可能会出现段错误。
33     // 当name指向的空间被释放之后, 如果再次被分配,
34     // 程序的运行结果就不可预知。
35     #if 0
36         // 没有将name指向NULL, 依然可以通过name访问对应的
37         // 内存空间, 并且运行程序不会报错
38         strcpy(name, "lisi");

```

```

39     printf("name = %s\n", name);
40 #else
41     // 如果将name指向NULL,再次对name进行操作,
42     // 编译不会报错,运行程序会出现段错误,
43     name = NULL;
44     strcpy(name, "lisi");
45     printf("name = %s\n", name);
46
47 #endif
48     return 0;
49 }

```

```

1  goto的使用场合:
2  #include <stdio.h>
3  #include <string.h>
4  #include <stdlib.h>
5  int main(int argc, const char *argv[])
6  {
7      /*your code*/
8      int *i_p = NULL;
9      short *s_p = NULL;
10     char *c_p = NULL;
11
12     // 对以上三个指针变量分别分配堆区的空间
13
14     if ((i_p = (int *)malloc(sizeof(int) * 10)) == NULL)
15     {
16         printf("malloc int failed\n");
17         return -1;
18     }
19
20     if ((s_p = (short *)malloc(sizeof(short)*10)) == NULL)
21     {
22         printf("malloc short failed\n");
23         free(i_p);
24         i_p = NULL;
25         return -2;
26     }
27
28     if ((c_p = (char *)malloc(sizeof(char)*10))==NULL)
29     {
30         printf("malloc char failed");
31         free(s_p);
32         s_p = NULL;
33         free(i_p);
34         i_p = NULL;
35         return -3;
36     }
37
38     free(c_p);
39     c_p = NULL;
40     free(s_p);
41     s_p = NULL;
42     free(i_p);
43     i_p = NULL;

```

```

44     return 0;
45 }
46 -----
47
48 #include <stdio.h>
49 #include <string.h>
50 #include <stdlib.h>
51 int main(int argc, const char *argv[])
52 {
53     /*your code*/
54     int *i_p = NULL;
55     short *s_p = NULL;
56     char *c_p = NULL;
57     int retVal;
58     // 对以上三个指针变量分别分配堆区的空间
59
60     if ((i_p = (int *)malloc(sizeof(int) * 10)) == NULL)
61     {
62         printf("malloc int failed\n");
63         retVal = 1;
64         goto ERR1;
65     }
66
67     if ((s_p = (short *)malloc(sizeof(short)*10)) == NULL)
68     {
69         printf("malloc short failed\n");
70         retVal = 2;
71         goto ERR2;
72     }
73
74     if ((c_p = (char *)malloc(sizeof(char)*10))==NULL)
75     {
76         printf("malloc char failed");
77         retVal = 3;
78         goto ERR3;
79     }
80
81     free(c_p);
82     c_p = NULL;
83     free(s_p);
84     s_p = NULL;
85     free(i_p);
86     i_p = NULL;
87
88
89     return 0;
90 ERR3:
91     free(s_p);
92     s_p = NULL;
93 ERR2:
94     free(i_p);
95     i_p = NULL;
96 ERR1:
97     return -retVal;
98 }

```

```
1  练习题:
2      定义一个int *类型的指针变量, 使用malloc在堆区分配sizeof(int) * 10大小的空间,
3      使用从终端输入的方式对malloc分配的堆区空间初始化,
4      使用冒泡排序的方式对堆区空间的数据进行排序。
5  #include <stdio.h>
6  #include <string.h>
7  #include <stdlib.h>
8  #define LEN 10
9
10 int *malloc_space(int len)
11 {
12     int *p = (int *)malloc(sizeof(int) * LEN);
13     if (p == NULL)
14     {
15         printf("malloc failed!\n");
16         return NULL;
17     }
18     return p;
19 }
20
21 void init_malloc_space(int *arr, int len)
22 {
23     for (int i = 0; i < len; i++)
24     {
25         scanf("%d", &arr[i]);
26     }
27 }
28
29 void bubble_sort(int *arr, int len)
30 {
31     for (int i = 0; i < len - 1; i++)
32     {
33         for (int j = 0; j < len - 1 - i; j++)
34         {
35             if (*(arr + j) > *(arr + j + 1))
36             {
37                 int tmp;
38                 tmp = *(arr + j);
39                 *(arr + j) = *(arr + j + 1);
40                 *(arr + j + 1) = tmp;
41             }
42         }
43     }
44 }
45
46 void print(int *arr, int len)
47 {
48     for (int i = 0; i < len; i++)
49     {
50         printf("%d ", arr[i]);
51     }
52     putchar('\n');
53 }
54
55 void destroy_malloc_space(int **pp)
```

```

56 {
57     free(*pp);
58     *pp = NULL;
59 }
60 int main(int argc, const char *argv[])
61 {
62     /*your code*/
63     // 1. 定义指针变量, 并分配堆区空间
64     int *arr_p = malloc_space(LEN);
65     // 2. 初始化
66     init_malloc_space(arr_p, LEN);
67
68     // 3. 冒泡排序
69     bubble_sort(arr_p, LEN);
70     // 4. 打印排序之后的结果
71     print(arr_p, LEN);
72
73     // 5. 释放堆区空间
74     destroy_malloc_space(&arr_p);
75
76     return 0;
77 }

```

4、多文件的编程

4.1 多文件编程相关的概念

1 在实际开发过程中, 每个综合的项目都可分成很多个不同的功能模块,
2 不可能将所有的代码都写到一个.c文件中, 而是采用多文件编程的方式,
3 将具有不同功能的代码写到对应的.c文件中, 最后只需要在main函数对应的.c
4 文件中调用其他.c文件中定义的函数即可。

5
6 比如: 温湿度报警的项目(led, 蜂鸣器, 温湿度传感器)

```

7     main.c
8     led.c      ---> led.h
9     beep.c     ---> beep.h
10    temp-hum.c ---> temp_hum.h

```

11
12 .h 文件被称为头文件, 主要书写的函数的声明, 宏定义, 变量的声明
13 .c 文件被称为源文件, 主要书写的函数的定义, 变量的定义

14
15 头文件中必须先写防止头文件重复包含的机制。

```

16 #ifndef __头文件名大写_H__
17 #define __头文件名大写_H__
18     // 宏定义
19     // 变量的声明
20     // 函数的声明
21 #endif

```

22
23 解释: 当第一次包含头文件时, #ifndef判断"__头文件名大写_H__"
24 没有被定义, 则以下的代码有效, 然后使用#define定义
25 "__头文件名大写_H__".

26	当第二次再次包含此头文件时， <code>#ifndef</code> 判断" <code>__头文件名大写_H__</code> "
27	被定义了，则以下代码将无效。

4.2 编写一个多文件编程的代码

main.c文件

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5  // 使用""包含自己编写的头文件，可以在文件前添加对应的头文件的路径
6  #include "./cal.h"
7  int main(int argc, const char *argv[])
8  {
9      /*your code*/
10     printf("100 + 200 = %d\n", cal_func(100, 200, '+', add_func));
11     return 0;
12 }
```

cal.c文件

```
1  int add_func(int a, int b)
2  {
3      return (a+b);
4  }
5  int sub_func(int a, int b)
6  {
7      return (a-b);
8  }
9  int mul_func(int a, int b)
10 {
11     return (a*b);
12 }
13 int div_func(int a, int b)
14 {
15     return (a/b);
16 }
17
18 int cal_func(int a, int b, char oper, int (*func_p)(int a, int b))
19 {
20     int ret = 0;
21     switch(oper)
22     {
23     case '+':
24         ret = func_p(a, b);
25         break;
26     case '-':
27         ret = func_p(a, b);
28         break;
29     case '*':
30         ret = func_p(a, b);
31         break;
32     case '/':
```

```

33     ret = func_p(a, b);
34     break;
35 }
36 return ret;
37 }

```

cal.h文件

```

1  #ifndef __CAL_H__
2  #define __CAL_H__
3  int add_func(int a, int b);
4  int sub_func(int a, int b);
5  int mul_func(int a, int b);
6  int div_func(int a, int b);
7
8  int cal_func(int a, int b, char oper, int (*func_p)(int a, int b));
9
10
11 #endif // __CAL_H__

```

```

1  编译多文件的代码：
2  gcc main.c cal.c

```

1 练习题：将上节课的malloc的代码使用多文件编程实现。

main.c文件

```

1  #include <stdio.h>
2  #include "malloc.h"
3  int main(int argc, const char *argv[])
4  {
5      /*your code*/
6      // 1. 定义指针变量，并分配堆区空间
7      int *arr_p = malloc_space(LEN);
8      // 2. 初始化
9      init_malloc_space(arr_p, LEN);
10
11     // 3. 冒泡排序
12     bubble_sort(arr_p, LEN);
13     // 4. 打印排序之后的结果
14     print(arr_p, LEN);
15
16     // 5. 释放堆区空间
17     destroy_malloc_space(&arr_p);
18     return 0;
19 }

```

malloc.c文件

```

1  #include "malloc.h"
2
3  int *malloc_space(int len)

```



```

4  {
5      int *p = (int *)malloc(sizeof(int) * LEN);
6      if (p == NULL)
7      {
8          printf("malloc failed!\n");
9          return NULL;
10     }
11     return p;
12 }
13
14 void init_malloc_space(int *arr, int len)
15 {
16     for (int i = 0; i < len; i++)
17     {
18         scanf("%d", &arr[i]);
19     }
20 }
21
22 void bubble_sort(int *arr, int len)
23 {
24     for (int i = 0; i < len - 1; i++)
25     {
26         for (int j = 0; j < len - 1 - i; j++)
27         {
28             if (*(arr + j) > *(arr + j + 1))
29             {
30                 int tmp;
31                 tmp = *(arr + j);
32                 *(arr + j) = *(arr + j + 1);
33                 *(arr + j + 1) = tmp;
34             }
35         }
36     }
37 }
38
39 void print(int *arr, int len)
40 {
41     for (int i = 0; i < len; i++)
42     {
43         printf("%d ", arr[i]);
44     }
45     putchar('\n');
46 }
47
48 void destroy_malloc_space(int **pp)
49 {
50     free(*pp);
51     *pp = NULL;
52 }

```

malloc.h文件

```

1  #ifndef __MALLOC_H__
2  #define __MALLOC_H__
3

```

```

4  #include <stdio.h>
5  #include <stdlib.h>
6
7  #define LEN 10
8
9  int *malloc_space(int len);
10 void init_malloc_space(int *arr, int len);
11 void bubble_sort(int *arr, int len);
12 void print(int *arr, int len);
13 void destroy_malloc_space(int **pp);
14
15 #endif // __MALLOC_H__

```

5、6种存储类型

5.1 定义变量的格式

```

1  存储类型  数据类型  变量名 = 初始值;
2

```

5.2 6种存储类型

```

1  auto const extern static register volatile

```

5.3 auto存储类型 --> 自动类型

```

1  auto : 自动存储类型
2
3  非自动类型的变量：全局变量，使用static修饰的全局变量，
4      使用static修饰的局部变量都属于非自动类型的变量，不可以使用auto修饰。
5
6  自动类型的变量：局部变量属于自动类型的变量，可以使用auto进行修饰，
7      即使不加auto修饰，默认也是自动类型的变量，因此在开发中基本不使用auto。

```

```

1  #include <stdio.h>
2  // 全局变量，全局变量不属于自动存储类型的变量
3  // 不可以加auto进行修饰
4  // auto int global; // error
5  // auto static int global_s; // error
6
7
8  int main(int argc, const char *argv[])
9  {
10     /*your code*/
11     // 局部变量
12     auto int local = 0; // ok,省略auto一样属于自动存储类型的变量

```

```

13 // auto static int local_s; // error,
14
15 return 0;
16 }

```

5.4 const存储类型 --> 常量

```

1 const : 只读的变量
2 const可以修饰普通类型的变量，也可以修饰指针类型的变量。
3
4 const可以用来修饰函数的形参和返回值
5     int strlen(const char *s)
6     {
7         int len = 0;
8         while (*s != '\0') {
9             len++;
10        }
11        return len;
12    }
13    char *strcpy(char *s1, const char *s2);
14    char *strcat(char *s1, const char *s2);
15    int strcmp(const char *s1, const char *s2);

```

```

1 #include <stdio.h>
2 // 1. const 修饰全局变量
3 const int global = 1000;
4 // const修饰的全局变量存储在只读数据段(常量区)，
5 // 不可以被修改，即使通过指针也不可以被修改。
6 // 定义常量时，必须在定义的时候进行初始化。
7
8 int main(int argc, const char *argv[])
9 {
10     /*your code*/
11     // 2. const修饰局部变量
12     const int local = 2000;
13     // const修饰的局部变量在栈区分配空间，
14     // 不可以通过常量的变量名本身进行修改，
15     // 但是可以通过指针修改使用const修饰的局部变量。
16     // local = 3000;
17     int *p = &local;
18     printf("修改之前, local = %d\n", local);
19     *p = 3000; // 通过指针修改使用const修饰的局部变量。
20     printf("修改之后, local = %d\n", local);
21
22
23     // 3. const修饰指针变量
24     // const修饰*p1，不可以修改*p1指向的空间的内容
25     int a = 10086;
26     int b = 10010;
27     int const *p1 = &a;
28     // *p1 = 10010; // 不可以修改*p1指向的空间的内容
29     p1 = &b; // 可以修改p1的指向

```

```

30
31 // const修饰*p2 , 不可以修改*p指向的空间的内容
32 const int *p2 = &a;
33 // *p2 = 10010; // 不可以修改*p2指向的空间的内容
34 p2 = &b; // 可以修改p2的指向
35
36 // const修饰p3 , 不可以修改p3的指向
37 int * const p3 = &a;
38 *p3 = 10010; // 可以修改*p3指向的空间的内容
39 // p3 = &b; // 不可以修改p3的指向
40
41 // 第一个const修饰*p4, 不可以修改*p4指向的空间的内容
42 // 第二个const修饰P4, 不可以修改p4的指向
43 const int *const p4 = &a;
44 // *p4 = 10010; // 不可以修改*p3指向的空间的内容
45 // p4 = &b; // 不可以修改p3的指向
46 return 0;
47 }

```

5.5 extern存储类型 --> 外部的

- 1 extern : 外部的, 在其他文件中定义的全局变量或者函数如果想在当前文件中使用,
- 2 需要使用extern进行声明, 表示在其他文件中定义的。
- 3 1. 修饰全局变量: 表示此全局变量是在其他.c文件中定义的。
- 4
- 5 2. 修饰函数: 表示此函数是在其他.c文件中定义的。

main.c文件

```

1 #include <stdio.h>
2
3 // 在main函数中调用extern.c文件中定义的全局变量或者函数时,
4 // 需要使用extern进行声明
5 extern int global; // 注, 使用extern声明全局变量时, 不要进行赋值的操作
6 // extern int global = 10010; // 错误的
7
8 // 对于函数来说, 加不加extern效果是一样的。
9 extern void print(void);
10
11
12 int main(int argc, const char *argv[])
13 {
14     /*your code*/
15     printf("main.c::global = %d\n", global);
16     printf("main.c::&global = %p\n", &global);
17
18     print();
19     return 0;
20 }

```

extern.c文件

```

1 #include <stdio.h>
2 // 定义全局变量
3 int global = 10086;
4
5 // 定义函数
6 void print(void)
7 {
8     printf("extern.c::global = %d\n", global);
9     printf("extern.c::&global = %p\n", &global);
10 }

```

5.6 static存储类型 --> 静态

```

1 static : 静态的存储类型
2 1. 修饰全局变量: 不可以被外部的其他文件使用
3
4 2. 修饰函数: 不可以被外部的其他的文件使用
5
6 3. 修饰局部变量: 延长变量的生命周期到整个程序结束,
7     使用static修饰的局部变量如果初始化, 则在.data段分配空间;
8     如果使用static修饰的局部变量没有初始化, 则在.bss段分配空间,
9     并且初始化为0.
10    使用static修饰的局部变量, 只在函数第一次调用时被初始化一次,
11    后边再次调用此函数, 则不在执行初始化的代码

```

main.c文件

```

1 #include <stdio.h>
2 // 在main.c文件中不可以调用static.c文件中,
3 // 使用static修饰的全局变量和函数, 以下编译报错
4 // extern int global;
5 // extern void print(void);
6
7 // 使用static的好处, 可以在不同的文件中定义
8 // 全局变量名和函数名相同的变量和函数
9
10 // 定义的global变量和print函数和static.c文件的变量和函数
11 // 不是同一个变量和函数, 只是名字相同而已。
12 int global = 10086;
13
14 void print(void)
15 {
16     printf("main.c::global = %d\n", global);
17     printf("main.c::&global = %p\n", &global);
18 }
19
20 // static修饰局部变量, 延长局部变量的生命周期
21 int auto_add(void)
22 {
23     // 变量i, 在.data段分配空间
24     static int i = 100; // 只在第一次调用此函数时, 对i进行赋值操作
25     i++;

```

```

26     return i;
27 }
28 int auto_sub(void)
29 {
30     // 变量i, 在.bss段分配空间, 默认初始化为0
31     static int i;
32     i--;
33     return i;
34 }
35 int main(int argc, const char *argv[])
36 {
37     /*your code*/
38     // 使用的时main.c文件中的global变量和print函数,
39     // 而不是使用的static.c文件中的global变量和print函数
40     printf("main.c::global = %d\n", global);
41     printf("main.c::&global = %p\n", &global);
42     print();
43
44     int a = 0;
45     a=auto_add();    // 101
46     a=auto_add();    // 102
47     a=auto_add();    // 103
48     a=auto_add();    // 104
49     printf("a = %d\n", a); // 104
50
51     int b = 0;
52     b=auto_sub();    // -1
53     b=auto_sub();    // -2
54     b=auto_sub();    // -3
55     b=auto_sub();    // -4
56     printf("b = %d\n", b); // -4
57     return 0;
58 }

```

static.c文件

```

1  #include <stdio.h>
2
3  // 定义全局变量, 和函数使用static修饰,
4  // 使用static修饰的函数不可以被其他文件调用
5  static int global = 10086;
6
7  static void print(void)
8  {
9      printf("static.c::global = %d\n", global);
10     printf("static.c::&global = %p\n", &global);
11 }
12
13

```

5.7 register存储类型 --> 寄存器

```
1 1. register : 寄存器类型的存储变量
2
3 2. 定义寄存器类型的变量使用时, 和普通变量的用法一样。
4
5 3. 对于寄存器类型的变量, 不可以进行取地址的运算。
6     原因: 寄存器的访问是通过编号进行访问的, 没有地址。
7
8 4. 在实际开发中, 尽量不要过多的定义寄存器类型的变量。
9     原因: 寄存器的个数有限。
10
11 5. 定义寄存器类型的变量的读写速度比普通的变量的读写速度快。
12     原因: 寄存器的读写速度比内存的读写速度快。
13
14     疑问1: 为什么寄存器类型的变量不能取地址运算?
15     疑问2: 为什么寄存器的个数有限?
16     疑问3: 寄存器读写速度为什么快?
17     《ARM体系结构及接口技术》时, 详细的分析寄存器。
18
```

```
1 #include <stdio.h>
2
3 int main(int argc, const char *argv[])
4 {
5     /*your code*/
6     // 1. 定义寄存器类型的变量
7     register int r = 1000;
8     // 2. 使用寄存器类型的变量
9     printf("r = %d\n", r);
10    r = 2000;
11    printf("r = %d\n", r);
12
13    // 4. 不可以进行取地址的运算
14    // printf("&r = %p\n", &r); // error
15    return 0;
16 }
```

5.8 volatile存储类型 --> 易变的

```
1 volatile : 易变的
2 底层课程时讲解volatile关键字的使用。
```

6、作业

```
1 理解今天的代码, 自己动手写一遍。
2 C语言的试卷, 第7套, 前20题
```

7、明天的授课内容

- 1 1. **typedef** 重新定义新的数据类型
- 2 2. 结构体(结构体指针, 结构体数组, 结构体指针数组, 结构体数组指针)
- 3
- 4 学生成绩管理系统。