1、书接上回，函数

1.1 函数的形参

1.1.1 普通类型的形参(值传递)

```
1 int add_func(int a, int b)       // 使用值
  传递即可
2 {
3     return a+b;
4 }
5
6 int add_func(int *a, int *b)         //
  使用地址传递
7 {
8     return *a+*b;
9 }
```

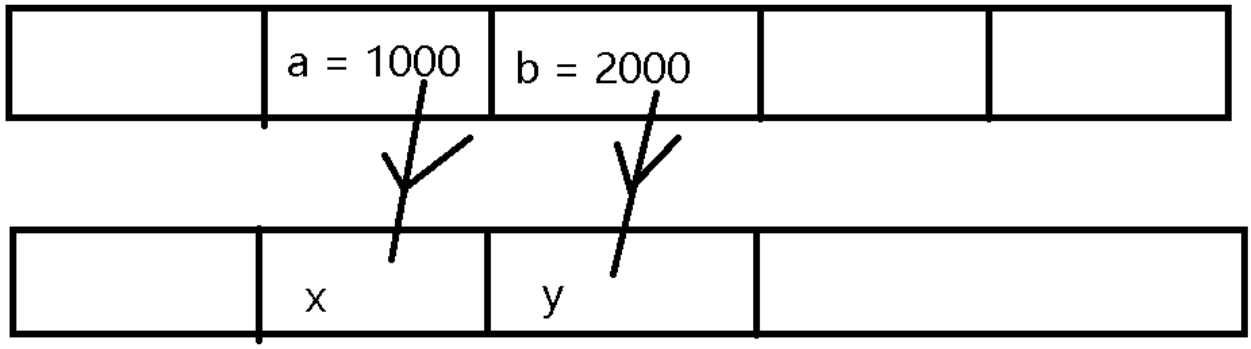
1.1.2 一级指针类型的形参(地址传递)

```
 1 实现一个函数，通过函数交互两个变量的值
 2 #include <stdio.h>
 3 void show(int a, int b)
 4 {
 5     printf("a = %d, b = %d\n", a, b);
 6 }
 7 void swap(int x, int y)
 8 {
 9     int tmp;
10     tmp = x;
11     x = y;
```

```
12      y = tmp;
13  }
14
15  void swap2(int *p, int *q)
16  {
17      *p = *p ^ *q;
18      *q = *p ^ *q;
19      *p = *p ^ *q;
20
21      // *p = 1000 1000
22      // *q = 0111 1010
23      // *p = *p ^ *q = 1111 0010
24      // *q = *p ^ *q = 1000 1000
25      // *p = *p ^ *q = 0111 1010
26  }
27
28  int main(int argc, const char *argv[])
29  {
30      int a = 1000, b = 2000;
31      // 编写函数实现a和b变量的值的交换
32      printf("交换之前 >");
33      show(a,b);
34      swap(a, b);    // 值传递，不可以进行交
    换，
35      printf("交换之后 >");
36      show(a, b);
37
38      printf("交换之前 >");
39      show(a,b);
40      swap2(&a, &b);    // 地址传递，可以进行
    交换，
```

```
41      printf("交换之后 >");
42      show(a, b);
43
44      return 0;
45  }
46
```

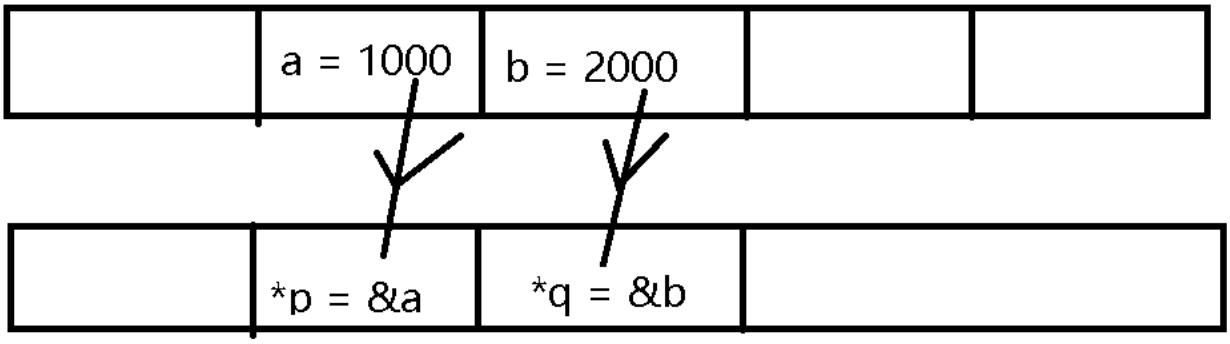
```
int main()    {
    int a = 1000, b = 2000;
    swap(a, b);  // 值传递，不可以进行交换，
}
```




```
void swap(int x, int y)
{
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

交换的是x和y变量对应内存空间的值。

```
int main()    {
      int a = 1000, b = 2000;
      swap(a, b);  // 值传递，不可以进行交换，
}
```




```
void swap2(int *p, int *q)
{
      *p = *p ^ *q;
      *q = *p ^ *q;          地址传递可以完成数
      *p = *p ^ *q;          据的交互
}
```

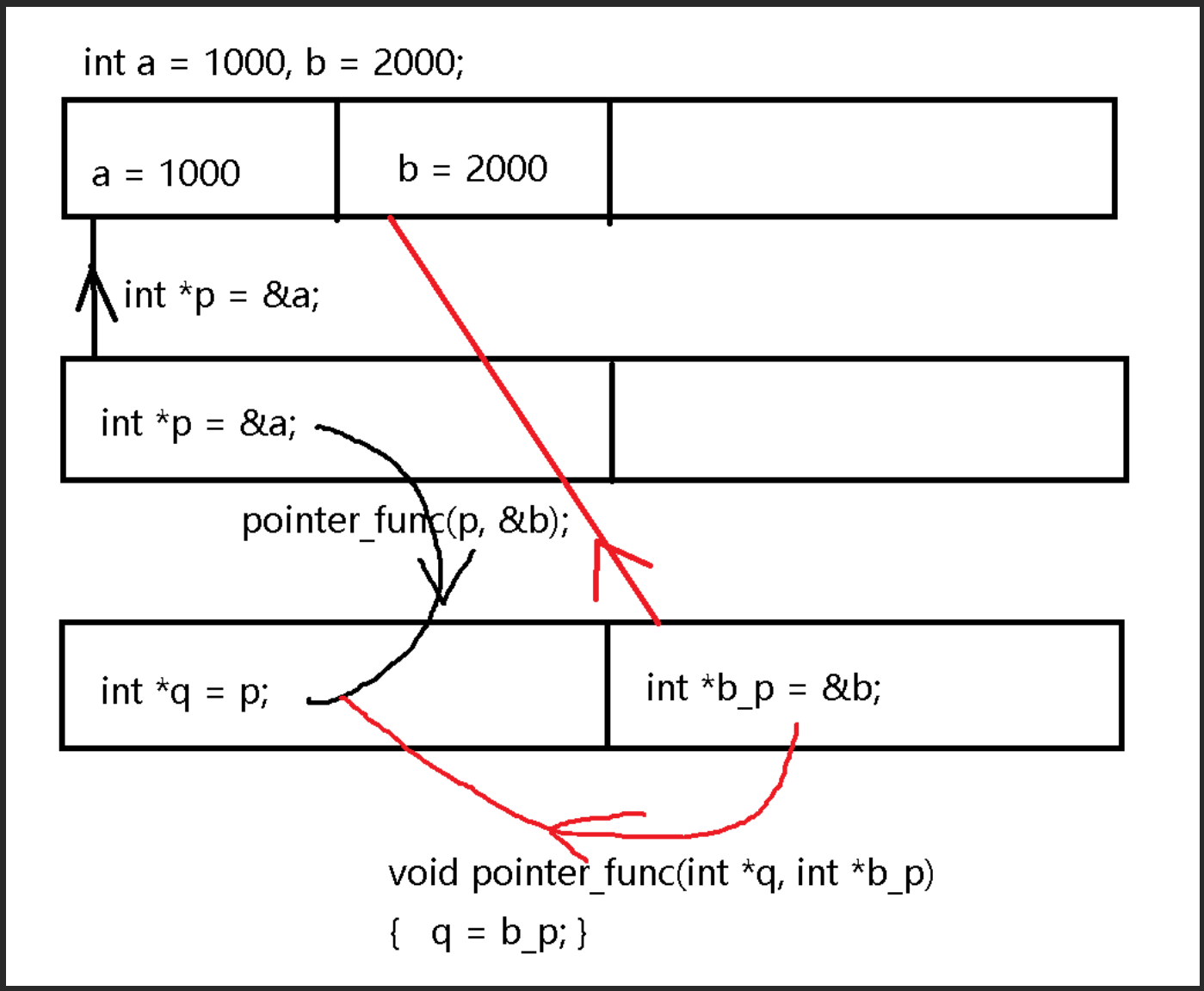
```c
定义函数，通过参数返回函数的结果
#include <stdio.h>
// a和b属于输入型的参数
int add_func(int a, int b)
{
      return a+b;
}

// a和b属于输入型的参数，val属于输出型的参数
void mul_func(int a, int b, int *val)
{
      *val =  a * b;
}
```

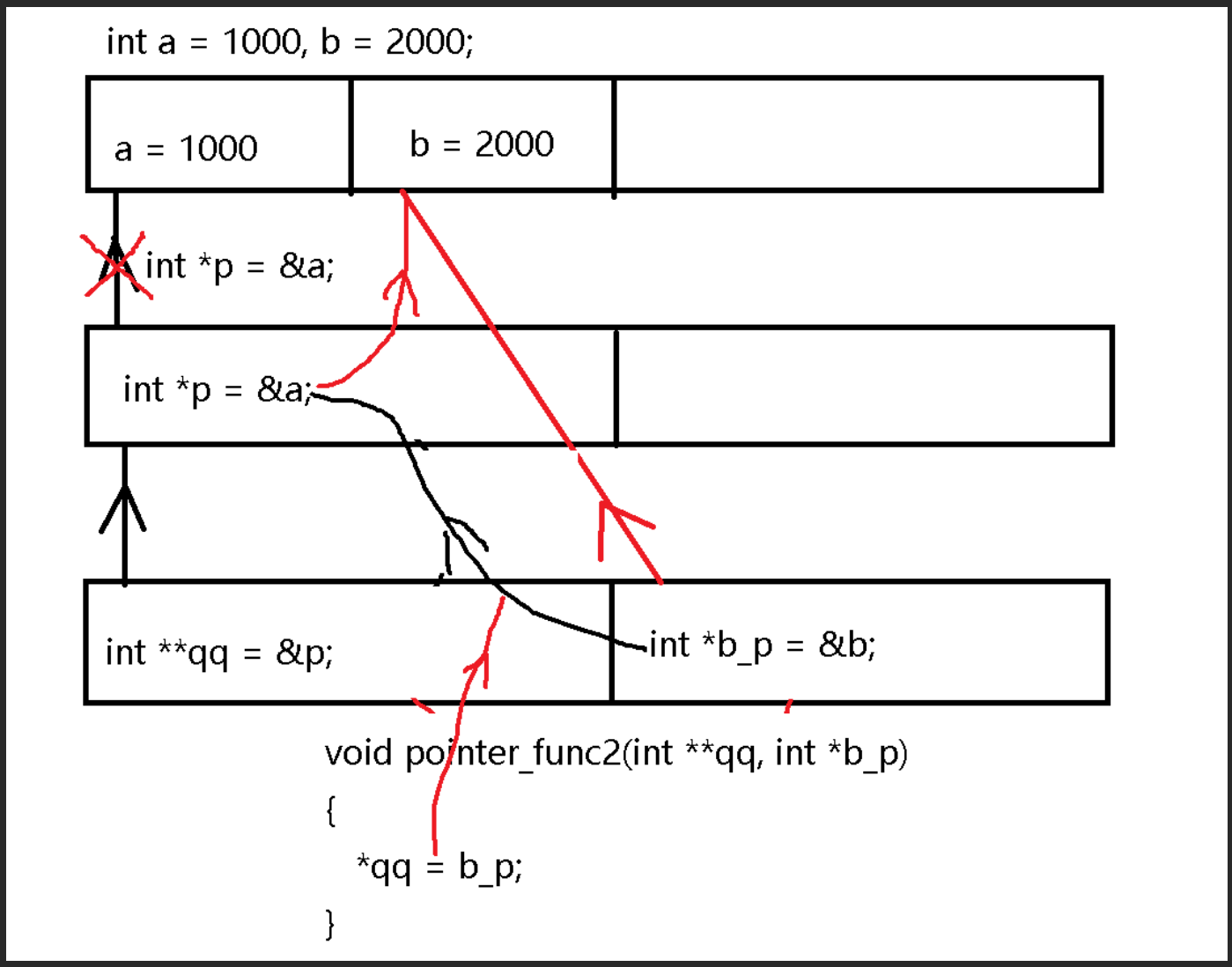
```
14
15 int main(int argc, const char *argv[])
16 {
17     int sum = add_func(100,200);
18     printf("sum = %d\n", sum);
19
20     int mul;
21     mul_func(100,200,&mul);
22     printf("mul = %d\n", mul);
23     return 0;
24 }
25
```

1.1.3 二级指针类型的形参(地址传递)

```
1 1 给函数传递一个指针数组类型的参数，可以使用二级指针。
2     参考二级指针和指针数组的关系案例
3
4 2. 通过函数修改一级指针的指向
5 #include <stdio.h>
6 void pointer_func(int *q, int *b_p)
7 {
8     q = b_p;
9 }
10 void pointer_func2(int **qq, int *b_p)
11 {
12     *qq = b_p;
```

```
13 }
14 int main(int argc, const char *argv[])
15 {
16     int a = 1000, b = 2000;
17     printf("&a = %p\n &b = %p\n", &a,
   &b);
18     int *p = &a;
19     // p = &b; // 修改指针变量p的指向
20     printf("修改p的指向之前 = %p\n", p);
21     pointer_func(p, &b);
22     printf("修改p的指向之后 = %p\n", p);
23
24     printf("修改p的指向之前 = %p\n", p);
25     pointer_func2(&p, &b);
26     printf("修改p的指向之后 = %p\n", p);
27
28     return 0;
29 }
30
```


int a = 1000, b = 2000;
a = 1000
b = 2000
int *p = &a;
int *p = &a;
pointer_func(p, &b);
int *q = p;
int *b_p = &b;
void pointer_func(int *q, int *b_p)
{ q = b_p; }



1.2 一维数组作为函数的参数(一级指针)

```
1 void function(int arr[], int len);
2 void function(int *arr_p, int len);
3     参考一维数组和一级指针关系的代码
```

1.3 二维数组作为函数的参数(数组指针)

```
1 void function(int arr[][列宽], int row,
  int col);
2 void function(int (*arr_p)[列宽], int
  row, int col);
3     参考二维数组和数组指针关系的代码
```

1.4 指针数组作为函数参数(二级指针)

```
1 void function(int *arr[], int len);
2 void function(int **p_arr, int len);
3 void function(char *arr[], int len);
4 void function(char **p_arr, int len);
5     参考指针数组和二级指针关系的代码
```

1.5 main函数的参数

```
1 /*
2  * 功能：每个程序有且只有一个main函数，程序的
  入口函数
3  * 参数：
4  *       @ argc : 表示执行程序时，传递的参数
  的个数，
5  *              包括可执行程序的名字
6  *              比如：  ./a.out  hello  world
  argc = 3
7  *       @ argv : 字符指针数组，每个成员都是
  char *类型
```

```
 *              字符指针数组中每个成员存放的都是一个地址,
 *              字符指针数组的每个成员存放的是字符串的首地址,
 *              argv[0] = "./a.out"      字符串的首地址
 *              argv[1] = "hello"
 *              argv[2] = "world"
 * 返回值:
 *      int类型, 成功返回0;
 *      失败返回一个非0
*/
int main(int argc, const char *argv[])
{
    return 0;
}

```

```
#include <stdio.h>
int main(int argc, const char *argv[])
{
    if (argc != 3)
    {
        printf("执行程序时, 传递的参数不合理, 请重新执行!\n");
        printf("usage:%s s1  s2\n", argv[0]);
        return -1;
    }

    // 打印执行脚本文件时传递的所有的参数
```

```
    for (int i = 0; i < argc; i++)
    {
        printf("argv[%d] = %s\n", i, argv[i]);
    }

    return 0;
}

```

```
执行程序时，传递参数，完成一个简单的计算器。
./cal  100 + 200          // OK，执行程序时，传递3个参数

./cal  100+200            // OK，执行程序时，传递1个参数

gcc 03cal.c -o cal
```

```
#include <stdlib.h>

int atoi(const char *nptr);
/*
 *  功能：将整型字符串转换为整型
 *  参数：
 *      @ nptr : 整数的字符串，比如："12345"
 *  返回值：
 *      字符串对应的整数值，比如：12345
*/
```

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

int cal(int a, int b, char operator)
{
    int result;
    switch(operator) {
    case '+':
        result = a + b;
        break;
    case '-':
        result = a - b;
        break;
    case '*':
        result = a * b;
        break;
    case '/':
        result = a / b;
        break;
    case '%':
        result = a % b;
        break;
    }

    return result;
}

int main(int argc, const char *argv[])
{
    if (argc != 4)
    {
```

```
32          printf("执行程序时，传递的参数错误，请重新执行!\n");
33          printf("usage : %s lValue operator rValue\n", argv[0]);
34          return -1;
35      }
36
37      // atoi : 将整数的字符串转换为整数
38      int lValue = atoi(argv[1]);
39      int rValue = atoi(argv[3]);
40
41      printf("%s %s %s = %d\n",argv[1],argv[2],argv[3],
42                  cal(lValue, rValue, *argv[2]));
43      return 0;
44  }
45
```

1.6 指针函数

```
1. 指针函数:
    本质是一个函数，函数的返回类型为指针类型。

2. 格式:
    返回类型 * 函数名(函数的形参列表)
    {
        函数体;
    }
3. 注意事项
```

```
10      1> 不可以返回局部变量的地址，原因是函数执行结束，
11              局部变量被系统回收，就会出现非法空间的访问。
12      2> 可以返回全局变量的地址，原因是全局变量在程序结束之后才会被回收。
13      3> 可以返回函数的参数传递的地址，
14          比如：char *strcpy(char *s1, const char *s2){}
15              char *strcat(char *s1, const char *s2){}
```

```c
#include <stdio.h>
// 1. 定义指针函数，函数返回值为指针类型
int *add_func(int a, int b, int *sum_p)
{
    *sum_p = a + b;
    return sum_p;
}
int main(int argc, const char *argv[])
{
    int sum;
    // int *p = add_func(100, 200, &sum);
    // printf("sum = %d\n", *p);
    // 参数：&sum ：表示通过函数修改sum变量的值
    // 返回值：  指针类型，返回的是sum的地址
    // *add_func(100, 200, &sum) ：  取sum变量中的内容。
```

```
16      printf("sum = %d\n", *add_func(100, 200, &sum));
17
18      // 参数: &sum : 表示通过函数修改sum变量的值
19      add_func(1000, 2000, &sum);
20      // 通过变量名sum访问, 变量对应的内存中存储的数据
21      printf("sum = %d\n", sum);
22
23
24      return 0;
25  }
26
```

1.7 函数指针

```
1.  函数指针:
2       本质是一个指针类型的变量, 指向的是一个具有相同的参数, 相同的返回值的函数。
3       函数的名字可以表示函数的入口地址
4
5   2.  定义函数指针变量
6       返回类型  (*函数指针变量名)(形参列表);
7               // 函数指针变量, 变量名为: “函数指针变量名”
8
9   3.  对函数指针变量进行初始化
10      1> 定义函数指针变量的同时进行初始化,
```

```
11          返回类型  (*函数指针变量名)(形参列表) = 函数名;
12      2> 先定义函数指针变量，后进行初始化
13          返回类型  (*函数指针变量名)(形参列表);
14          函数指针变量名 = 函数名;
15
16  4. 函数指针变量的使用：将函数指针的变量名当成函数名使用。
17
18  5. 函数指针变量的使用:常用于函数的参数，回调函数。
```

```c
#include <stdio.h>

void show(char * str)
{
    printf("%s\n", str);
}

void print(void)
{
    printf("hello world");
}
int main(int argc, const char *argv[])
{
    // 定义一个函数指针类型的变量，指向show函数
    // show_p函数指针变量，需要指向一个函数的参数为char *类型，
    // 函数返回值为void类型的函数。
```

```
    void (*show_p) (char *str) = show;

    // show_p = print;        // 错误

    // 函数指针变量的使用，将函数指针变量名当成函数名使用
    show("hello world");
    // 函数指针变量的使用，就当前一个函数名使用即可
    show_p("hello world");
    return 0;
}

```

函数指针作为函数的形参，实现回调函数的功能：

```
#include <stdio.h>
#include <stdlib.h>
int add_func(int l, int r)
{
    return l + r;
}
int sub_func(int l, int r)
{
    return l - r;
}
int mul_func(int l, int r)
{
    return l * r;
}
int div_func(int l, int r)
{
```

```c
    return l / r;
}
int mol_func(int l, int r)
{
    return l % r;
}

// 函数指针类型的变量作为函数的参数，实现回调函数的功能
/*
 * 功能：计算器的函数
 * 参数：a : 左操作数    b : 右操作数
 *       cal_p : 进行的算数运算对应的函数指针
 * 返回值，计算的结果
*/
int cal_func(int a, int b, int (*cal_p)(int l, int r))
{
    return cal_p(a, b);
}

int main(int argc, const char *argv[])
{
    if (argc != 4)
    {
        printf("执行程序时传递的参数不合理，请重新执行！\n");
        printf("usage : %s lvalue operator rvalue\n", argv[0]);
        return -1;
    }
```

```
45
46      int l = atoi(argv[1]);
47      int r = atoi(argv[3]);
48      int result;
49      switch(*argv[2])
50      {
51      case '+':
52          result = cal_func(l, r,
    add_func);
53          break;
54      case '-':
55          result = cal_func(l, r,
    sub_func);
56          break;
57      case '*':
58          result = cal_func(l, r,
    mul_func);
59          break;
60      case '/':
61          result = cal_func(l, r,
    div_func);
62          break;
63      case '%':
64          result = cal_func(l, r,
    mol_func);
65          break;
66
67      }
68
69      printf("%s %s %s = %d\n", argv[1],
    argv[2], argv[3], result);
```

```
70
71
72      return 0;
73 }
74
```

1.8 函数指针数组

```
1 1. 函数指针数组 :
2       本质是一个数组, 数组中的每个成员都是一个函数指针类型的成员。
3
4 2. 格式:
5       数据类型 (*函数指针数组名[元素个数])(参数列表);
6
7 3. 将函数指针数组的每个元素当成函数名使用即可。
8       函数指针数组名[下标](实参列表);
9       (*(函数指针数组名 + 下标))(实参列表);
```

```c
#include <stdio.h>
int add_func(int l, int r)
{
    return l + r;
}
int sum_func(int l, int r)
{
    return l - r;
}

```

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

    // 定义一个函数指针数组，数组的每个成员都是函数指针
    // int (*函数名)(int, int);
    // int (*func_p_array[2])(int, int) = {add_func,sum_func};
    int (*func_p_array[2])(int, int) = {0};
    func_p_array[0] = add_func;
    func_p_array[1] = sum_func;

    int a = 1000;
    int b = 2000;
    int result;
    // 使用函数指针数组中的成员
    // 1. 将函数指针数组的每个元素当成一个函数名使用
    result = func_p_array[0](a, b);
    printf(" 1000 + 2000 = %d\n", result);
    result = func_p_array[1](a, b);
    printf(" 1000 - 2000 = %d\n", result);

    // 2. 将函数指针数组名当成一个指针使用，通过地址偏移的方式，
    // 访问函数指针数组的每个成员。
    result = (*(func_p_array + 0))(b, a);
```

```
34      printf(" 2000 + 1000 = %d\n",
   result);
35      result = (*(func_p_array + 1))(b,
   a);
36      printf(" 2000 - 1000 = %d\n",
   result);
37
38
39      return 0;
40  }
41
```

1.9 函数指针数组指针

```
1  1. 函数指针数组指针
2       本质：是一个指针，指向的是函数指针数组
3
4  2. 格式
5       int (*(*函数指针数组指针变量名))(参数列
   表);
```

```
 1  #include <stdio.h>
 2  int add_func(int l, int r)
 3  {
 4      return l + r;
 5  }
 6  int sum_func(int l, int r)
 7  {
 8      return l - r;
 9  }
```

```c

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

    // 1. 定义一个函数指针数组，数组的每个成员都是函数指针
    // int (*函数名)(int, int);
    // int (*func_p_array[2])(int, int) = {add_func,sum_func};
    int (*func_p_array[2])(int, int) = {0};
    func_p_array[0] = add_func;
    func_p_array[1] = sum_func;

    // 2. 定义函数指针数组指针，指向函数指针数组
    int(*(*func_p_array_p))(int, int) = func_p_array;

    int a = 10000, b = 20000, result;
    // 3. 函数指针数组指针的使用
    // 3.1 将函数指针数组指针变量当成数组的名字使用
    result = func_p_array_p[0](a, b);
    printf("10000 + 20000 = %d\n", result);
    result = func_p_array_p[1](a, b);
    printf("10000 - 20000 = %d\n", result);

```

```
32      // 3.2 将函数指针数组指针当成指针使用，通过地址偏移
33      result = (*(func_p_array_p + 0))(b, a);
34      printf("20000 + 10000 = %d\n", result);
35      result = (*(func_p_array_p + 1))(b, a);
36      printf("20000 - 10000 = %d\n", result);
37      return 0;
38  }
39
```