1、书接上回，指针

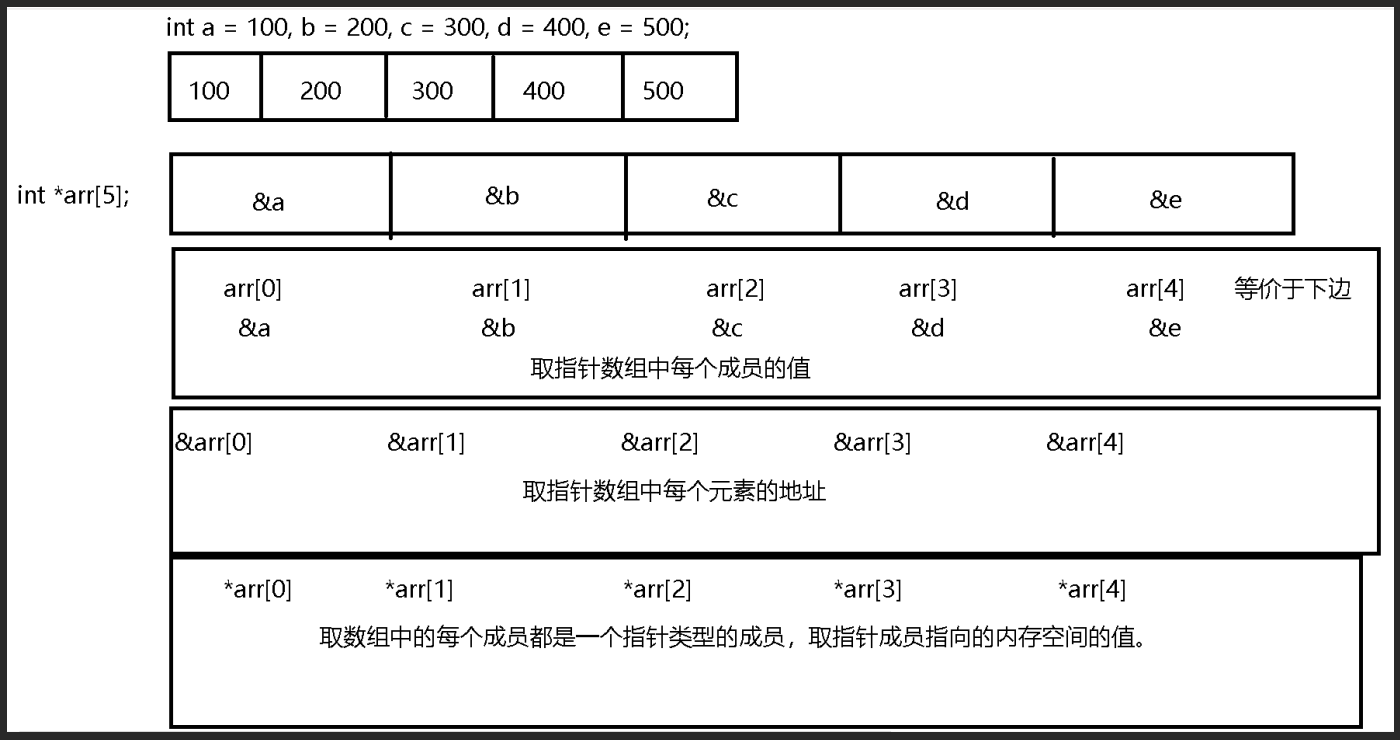
1.1 指针数组

```
1. 定义指针数组的格式
    数据类型 * 数组名[数组成员个数];

2. 特征
    1> 指针数组本质是一个数组,
    2> 数组的每个成员是一个"数据类型 *"指针类型,
        及数组的每个成员存放的都是一个地址(指针)。
```

```c
#include <stdio.h>
int main(int argc, const char *argv[])
{
    // 1. 定义指针数组, 并进行初始化
    int a = 100, b = 200, c = 300, d = 400, e = 500;
    // 初始化时给每个元素存储一个int *类型的指针(地址)。
    int *p_arr[5] = {&a, &b, &c, &d, &e};

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

```
10      printf("p_arr 成员个数 = %ld\n", sizeof(p_arr)/sizeof(int *));
11
12
13      // 将指针数组当成一个普通的一维数组看待
14      for (int i = 0; i < 5; i++)
15      {
16          // 访问指针数组每个成员的值，指针数组成员的值是一个地址
17          printf("p_arr[%d] = %p\n", i, p_arr[i]);
18      }
19      printf("-----------------------------\n");
20      for (int i = 0; i < 5; i++)
21      {
22          // 打印指针数组每个元素的地址，数组的每个成员都是指针类型的变量
23          printf("&p_arr[%d] = %p\n", i, &p_arr[i]);
24
25      }
26      printf("-----------------------------\n");
27      for (int i = 0; i < 5; i++)
28      {
29          // 访问指针数组每个成员指向的地址空间的内容
30          printf("*p_arr[%d] = %d\n", i, *p_arr[i]);
31      }
```

```
32
33      return 0;
34  }
35
```



1.2 数组指针

```
1. 定义数组指针的格式
    数据类型 (* 数组指针变量名)[列宽];

2. 特征
    1> 数组指针本质是一个指针
    2> 数组指针指向的是一个二维数组的列的个数等于"列宽"大小的二维数组。
    int arr[2][2] = {0};
    int (* p_arr)[2] = arr;  //  数组指针指向有两列的二维数组中
    int (*p_arr)[3] = arr;  // 错误 数组指针指向有三列的二维数组中
    3> 如果给函数传递一个二维数组,
        函数的形参的写法:
            第一种: 返回类型 函数名(数据类型 二维数组名[][列宽]) {}
            第二种: 返回类型 函数名(数据类型 (*数组指针变量名)[列宽]) {}
```

```
面试题: 数组指针和指针数组的区别?
    数组指针是一个指针类型变量, 指向的是一个二维数组;
    指针数组是一个数组, 数组的每个成员是一个指针类型。
```

1.2.1 研究二维数组名的作用

```c
#include <stdio.h>

int main(int argc, const char *argv[])
{
    // 1. 定义一个二维数组
    int arr[3][4] = {{1,2,3,4},
                     {5,6,7,8},
                     {9,10,11,12}};

    // 2. 探讨二维数组名
    //2.1 数组名表示：二维数组的首地址，是一个常量不可以被修改
    printf("arr 表示二维数组首地址 = %p\n", arr);
    // 二维数组的名字 ： 表示一个行地址(行指针)，+1表示加1行数组大小的空间
    printf("arr + 1 表示加一行元素大小的空间 = %p\n", arr + 1);

    // 2.2 数组名[行下标] ： 表示二维数组每一行的首地址
    printf("arr[1] 表示第一行的首地址 = %p\n", arr[1]);
    // 数组名[行下标] ：表示列地址，+1偏移一个数组成员大小的空间
    printf("arr[1] + 1  = %p\n", arr[1] + 1);

```

```c
    // 2.3 *二维数组名 ： 表示降维， 表示列地址(列指针)
    printf("*(arr + 2) 表示列地址 = %p\n", *(arr + 2));
    // *(arr + 2) 表示列地址， +1表示偏移一个数组成员大小的空间
    printf("*(arr + 2) + 1 表示列地址 = %p\n", *(arr + 2) + 1);

    // 2.4 *(*(二维数组名 + 行偏移) + 列偏移) : 取二维数组中的内容
    printf("arr[1][1] = %d\n", *(*(arr + 1) + 1));

    // *(二维数组名[行下标] + 列偏移) : 取二维数组中的内容
    printf("arr[1][1] = %d\n", *(arr[1] + 1));

    // 总结:
    // int arr[3][4] = {0};
    // arr : 二维数组的名，表示二维数组的首地址，表示行地址
    // arr + 1 : 偏移一行元素大小的空间，表示行地址
    // *(arr + 1) : 偏移一行之后，取*降维，表示列地址
    // *(arr + 1) + 1 : 偏移一行之后，取*降维，加1表示偏移一个数组成员大小的空间
    // arr[0]  : 表示第0行的首地址，表示列地址
```

```
39      //              等价于 *(arr + 0)
40      // *(*(arr + 1) + 1):取第一行，第一列成员中的内容
41      // *(arr[1] + 1) :  取第一行，第一列成员中的内容
42      //              等价于 *(*(arr + 1) + 1)
43
44      // 访问二维数组中每个成员的值
45      // 1. 通过数组名[行下标][列下标]
46      for (int i = 0; i < 3; i++)
47      {
48          for (int j = 0;j < 4; j++)
49          {
50              printf("%d ", arr[i][j]);
51          }
52          putchar('\n');
53      }
54      printf("----------------------\n");
55
56      // 2. 数组名加偏移量的方式
57      for (int i = 0; i < 3;i++)
58      {
59          for (int j = 0; j < 4; j++)
60          {
61              printf("%d ", *(*(arr + i) + j));
62          }
63          putchar('\n');
64      }
65      printf("----------------------\n");
66      for (int i = 0; i < 3;i++)
```

```
67      {
68          for (int j = 0; j < 4; j++)
69          {
70              printf("%d ", *(arr[i] +
    j));
71          }
72          putchar('\n');
73      }
74      printf("----------------------\n");
75      return 0;
76  }
77
```

1.2.2 数组指针和二维数组的关系

```
1       数组指针变量指向的是一个二维数组，将二维数
    组名赋值给数组指针变量。
2       而二维数组数组的名字是一个行地址，数组指针
    变量名也是一个行地址，
3       可以将数组指针变量名当成一个二维数组名使
    用。二维数组名怎样使用则
4       数组指针变量名也怎么使用即可。
5
6       区别：二维数组名是一个常量不可以被修改。
7       数组指针变量名是一个变量，可以被修改
```

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

```c
int main(int argc, const char *argv[])
{
    // 1. 定义一个二维数组
    int arr[3][4] = {{1,2,3,4},
                     {5,6,7,8},
                     {9,10,11,12}};

    // 2. 定义一个数组指针变量指向一个3行4列的二维数组
    int (*arr_p)[4] = arr;

    // 总结:
    // arr_p : 表示行地址
    // arr_p + 1 : 表示偏移一行
    // arr_p[0]  : 表示列地址, 表示第0行的首地址
    // *(arr_p + 1) : 表示列地址, 表示第1行的首地址
    // *(arr_p + 1) + 1 : 表示列地址, 表示第1行第1列元素的地址
    // arr_p[1] + 1 : 表示列地址, 表示第1行第1列元素的地址
    // *(*(arr_p + 1) + 1) : 表示第1行第1列元素的值
    // *(arr_p[1] + 1) : 表示第1行第1列元素的值

    // 通过数组指针指向二维数组, 然后访问二维数组中的所有的成员
    // 1. 将数组指针当成一个二维数组名使用即可
```

```
25      for(int i = 0; i < 3; i++)
26      {
27          for (int j = 0; j <4; j++)
28          {
29              printf("%d ", arr_p[i][j]);
30          }
31          putchar('\n');
32      }
33      printf("----------------------------
    \n");
34      // 2. 使用地址偏移的方式访问数组中的所有
    的成员
35      for(int i = 0; i < 3; i++)
36      {
37          for (int j = 0; j <4; j++)
38          {
39              printf("%d ", *(*(arr_p +
    i) + j));
40          }
41          putchar('\n');
42      }
43      printf("----------------------------
    \n");
44      for(int i = 0; i < 3; i++)
45      {
46          for (int j = 0; j <4; j++)
47          {
48              printf("%d ", *((arr_p[i])
    + j));
49          }
50          putchar('\n');
```

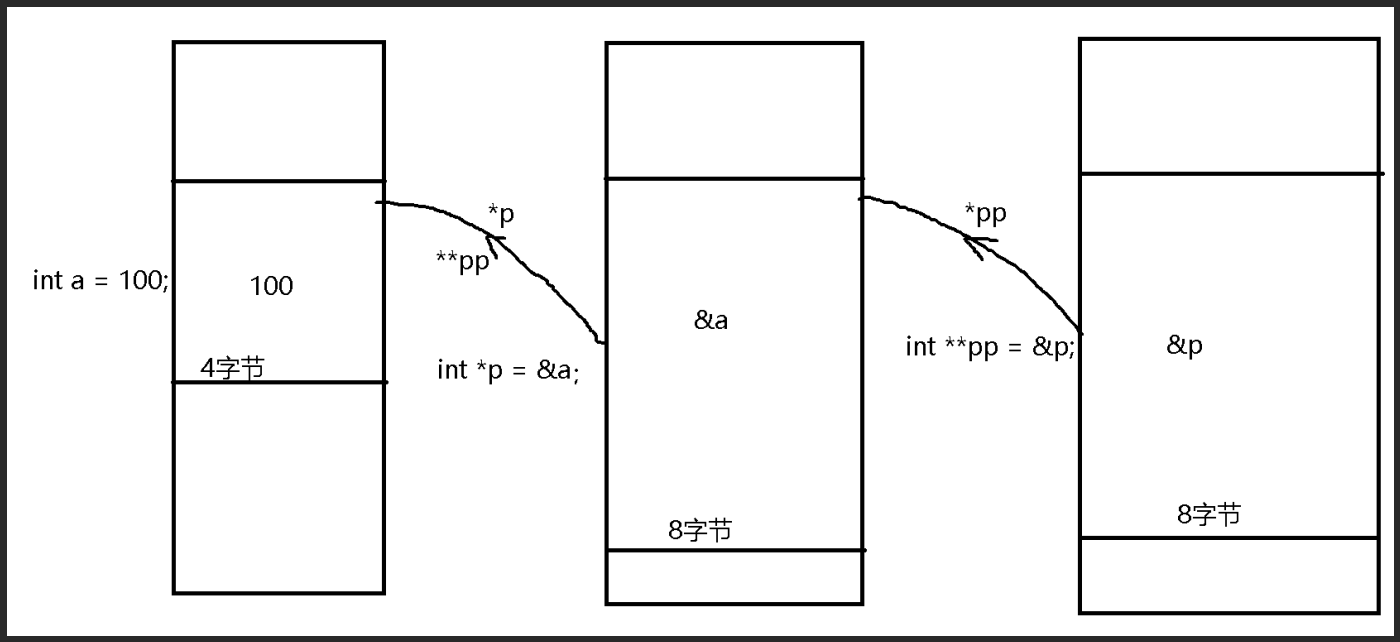
```
    }
    printf("--------------------------\n");

    // 总结:  二维数组和数组指针关系
    // arr <==> arr_p        : 行地址
    // arr + i <==> arr_p + i  : 行地址
    // *(arr + i) <==> arr[i] <==> *(arr_p + i) <==> arr_p[i] : 列地址
    // *(arr + i) + j <==> arr[i] + j <==> *(arr_p + i) + j <==> arr_p[i] + j : 列地址
    // arr[i][j] <==> *(*(arr + i) + j) <==> *(arr[i] + j) <==>
    //       arr_p[i][j] <==> *(*(arr_p + i) + j) <==> *(arr_p[i] + j)  : 取数组中每个元素的内容
    return 0;
}

```

```
练习题: 定义一个二维数组, 封装一个打印二维数组中所有的成员的函数,
    函数的参数为数组指针类型。
    void print_arr(int arr[][4], int row, int col);
    void print_arr(int (*arr_p)[4], int row, int col);

#include <stdio.h>

```

```c
void print_array(int (*arr_p)[4], int row, int col)
{
    for (int i = 0; i < row; i++)
    {
        for (int j = 0; j < col; j++)
        {
            printf("%d\t", *(*(arr_p + i) + j));
        }
        putchar ('\n');
    }
}

int main(int argc, const char *argv[])
{
    int arr[3][4] = {{1,2,3,4},
                     {5,6,7,8},
                     {9,10,11,12}};
    int row = sizeof(arr)/sizeof(arr[0]);
    int col = sizeof(*arr)/sizeof(int);
    print_array(arr, row, col);
    return 0;
}

```

1.3 二级指针

```
1. 定义二级指针的格式
    数据类型 **二级指针变量名;

2. 特征:
    二级指针变量中存放的的一级指针变量的地址(对一级指针变量取地址)。
    int a = 100;
    int *p = &a;
    int **pp = &p;

    等价关系:
    a <==> *p <==> **pp;    --> 取变量a地址空间的内存
    &a <==> p <==> *pp;     --> 一级指针, 一级指针的地址
            &p <==> pp;     --> 二级指针, 二级指针的地址
```




```c
#include <stdio.h>

```

```c
int main(int argc, const char *argv[])
{
    int a = 100;
    int *p = &a;
    // 二级指针的定义及初始化，需要对一级指针变量取地址
    int **pp = &p;

    // 修改变量a中的值
    a = 200;     // 通过变量本身进行修改
    printf("a = %d\n", a);
    *p = 300;    // 通过一级指针变量进行修改
    printf("a = %d\n", a);
    **pp= 400; // 通过二级指针变量进行修改
    printf("a = %d\n", a);



    //  a <==> *p <==> **pp;   --> 取变量a地址空间的内存
    printf("a = %d, *p = %d, **pp = %d\n", a, *p, **pp);
    //  &a <==> p <==> *pp;     --> 一级指针，一级指针的地址
    printf("&a = %p,\n p = %p,\n *pp = %p\n", &a, p, *pp);
    //          &p <==> pp;      --> 二级指针，二级指针的地址
    printf("&p = %p,\n pp = %p\n", &p, pp);

```

```
27      int b = 2000;
28      int *q = &b;
29      pp = &q;      // 修改二级指针的指向，修改的是二级指针变量中的值(一级指针的地址)
30
31      return 0;
32  }
33
```

1.4 二级指针和指针数组的关系

```
1  探讨几个问题：
2  int a = 100;
3  int *p = &a;      // p : 一级指针，+1偏移int类型大小的空间
4                    // *p : 取指向对应空间中的内存，即取变量a的值
5  int **pp = &p;    // pp : 二级指针，  +1偏移int *类型大小的空间
6                    // *pp : 一级指针，  +1偏移int类型大小的空间
7                    // **pp : 取指向对应空间中的内存，即取变量a的值
8
9  int *p_arr[5] = {&a, &b, &c, &d, &e};
10     // p_arr : 数组名，表示数组的首地址，数组每个成员都是int *类型的
11     //         数组名是一个常量，不可以被修改
```

```
    //  指针数组本质就是一个一维数组，数组的成员为指针类型。
    // p_arr ：指针数组的首地址，+1偏移int *类型的大小的空间
    // p_arr[下标] ：取指针数组中的每个成员的值，每个成员都是int *类型的指针
                    +1偏移int类型大小的空间
    // *(p_arr + 下标) ：取指针数组中的每个成员的值，每个成员都是int *类型的指针
                    +1偏移int类型大小的空间
    // *p_arr[下标] ：表示指针数组每个成员指向的地址空间的内容
    // **(p_arr + 下标) ：表示指针数组每个成员指向的地址空间的内容

如果给函数传递一个指针数组类型的实参时，可以使用二级指针接收指针数组类型的参数。
```

```c
#include <stdio.h>

int main(int argc, const char *argv[])
{
    // 验证二级指针
    int a = 100;
    int *p = &a;
    int **pp = &p;

    printf("pp = %p\n", pp); // 存放一级指针变量的地址
    printf("pp + 1 = %p\n", pp + 1); // 偏移int *类型的大小的空间
```

```c
12
13      printf("*pp = %p\n", *pp); // 存放普通变量的地址
14      printf("*pp + 1 = %p\n", *pp + 1); // 偏移int类型大小的空间
15
16      printf("**p = %d\n", **pp); // 取a变量中的内容
17      printf("**p + 1 = %d\n", **pp + 1); // 取出变量a的值之后然后加1;
18
19      // 指针数组
20      int  b = 200, c = 300, d = 400, e = 500;
21      int *p_arr[5] = {&a, &b, &c, &d, &e};
22      printf("p_arr = %p\n", p_arr);  // 表示数组的首地址
23      printf("p_arr + 1 = %p\n", p_arr + 1); // 偏移int *类型的大小
24
25      printf("p_arr[0] = %p\n", p_arr[0]); // 取出指针数组的第0个元素
26      printf("*(p_arr + 0) = %p\n", *(p_arr + 0)); // 取出指针数组的第0个元素
27
28      printf("*p_arr[1] = %d\n", *p_arr[1]);// 取b变量中的内容
29      printf("**(p_arr + 1) = %d\n", **(p_arr + 1)); // 取b变量中的内容
30      return 0;
```

```
}

```

```
定义一个二级指针，指向一个指针数组，通过二级指针访问指针数组中每个成员
指向的地址空间。

#include <stdio.h>
int main(int argc, const char *argv[])
{
    int a,b,c,d,e;
    int *p_arr[5] = {&a, &b, &c, &d, &e};
    // 定义二级指针，指向指针数组
    int **pp = p_arr;

    // 通过二级指针对指针数组中每个成员指向的空间进行初始化
    for (int i = 0; i < 5; i++)
    {
        **(p_arr + i) = (i + 1) * 1000;
    }

    // 将二级指针变量当成指针数组名使用
    for (int i = 0; i < 5; i++)
    {
        printf("*pp[%d] = %d ", i, *pp[i]);
    }
    printf("\n------------------------\n");
```

```
24
25      // 通过地址偏移的方式访问指针数组中每个成员指向的地址空间
26      for (int i = 0; i < 5; i++)
27      {
28          printf("**(pp + %d) = %d ", i, **(pp + i));
29      }
30      printf("\n-------------------------\n");
31
32      return 0;
33  }
34
```

```
1  定义一个打印指针数组中每个成员指向地址空间中内容的函数,
2  此函数的参数类型为二级指针类型。
3  void print_pointer_array(int *p_arr[], int len);
4  void print_pointer_array(int **pp, int len);
5
6  #include <stdio.h>
7  void init_pointer_array(int **pp, int len)
8  {
9      // 通过二级指针对指针数组中每个成员指向的空间进行初始化
10     for (int i = 0; i < len; i++)
11     {
```

```
12          **(pp + i) = (i + 1) * 1000;
13      }
14  }
15
16  void print_pointer_array(int **pp, int
    len)
17  {
18      // 通过地址偏移的方式访问指针数组中每个成
    员指向的地址空间
19      for (int i = 0; i < len; i++)
20      {
21          printf("**(pp + %d) = %d ", i,
    **(pp + i));
22      }
23  }
24
25  int main(int argc, const char *argv[])
26  {
27      int a,b,c,d,e;
28      int *p_arr[5] = {&a, &b, &c, &d,
    &e};
29
30      init_pointer_array(p_arr, 5);
31      print_pointer_array(p_arr, 5);
32
33      return 0;
34  }
35
```

1.5 内存的划分

```
1 32位系统的内存的划分：32位操作系统可以访问的虚拟地址空间位0-4G。
2 目前课上使用%p打印的地址都属于虚拟地址，操作系统为了更加安全，
3 在系统之上禁止操作物理地址内存的空间。
4
5 MMU:内存管理单元，这是一个芯片中的硬件，完成物理地址到虚拟地址的映射。
```

1.6 字符指针和字符串的关联

```
1 存储字符串的方式：
2
3 1. 使用一维字符数组，或者二维字符数组存储字符串：
```

```
    char name[20] = "zhangsan";
    char name[2][20] = {"zhangsan", "lisi"};
    使用字符数组存储字符串在栈区存储字符串。

    name = "lisi";  // 不可以，数组名为常量

2. 使用字符指针指向一个字符串常量
    char *str_p = "hello world";
              |           |-----> 这是一个字符串常量，在只读数据段存储
              |           |-----> 不可以通过指针变量修改字符串常量
              |---> 局部变量在栈区分配空间，存储的是字符串常量的地址

    strcpy(str_p, "nihao"); // 错误
    str_p = "nihao"; // 可以  修改字符指针变量指向另外一个字符串常量
```

```
3. 使用字符指针数组存储多个字符串常量的地址
    char *str_p_arr[2] = {"hello", "world"};
              |                 |----> 字符串常量区，及rodata段
              |----> 局部变量在栈区分配空间，每个成员都是字符指针类型
```

```c
#include <stdio.h>
#include <string.h>
void print_char_array(char *s)
{
    printf("%s\n", s);
}

void print_char_array2(char (*s)[20], int row)
{
    for (int i = 0; i < row; i++)
    {
        // 获取二维字符数组的每一行的首地址
        //printf("%s\n", (char *)(s + i));
        printf("%s\n", *(s + i));
    }
}
void print_char_pointer_array(char **str_pp, int len)
{
    for (int i =0; i < len; i++)
    {
        printf("%s\n", *(str_pp + i));
    }
}
int main(int argc, const char *argv[])
{

    // 回顾，字符数组存储字符串
    char name[20] = "zhangsan";
```

```
29      char name2[2][20] = {"busan", "busi"};
30      print_char_array(name);
31      print_char_array2(name2, 2);
32
33
34      // 使用字符指针指向一个字符串常量
35      char *str_p = "hello";
36      print_char_array(str_p);
37
38      // strcpy(str_p, "world"); // 编译不报错，运行包段错误
39      str_p = "world";    // 修改字符指针的指向
40      print_char_array(str_p);
41
42      char *str_q = "world";
43      // str_p和atr_q都指向了同一个字符串常量"world",
44      // 相同的字符串常量在rodata段只会存在一份。
45      printf("str_p = %p\n", str_p);
46      printf("str_q = %p\n", str_q);
47
48
49      // 定义字符指针数组，存储字符串常量
50      char *str_arr[2] = {"hello", "world"};
51      print_char_pointer_array(str_arr, 2);
52
```

```
53
54      return 0;
55  }
56
```

2、函数

2.1 函数概念

```
1       将具有特定功能的一段代码，封装成一个代码块，当使用此代码时，
2       可以通过调用的方式进行使用。
3       封装函数之后，需要重复被使用的代码不需要重复书写，直接通过
4       函数的调用实现即可。
5
6       比如：printf,scanf,putchar,getchar,puts,gets
7           算法库，
8
9       调用别人实现的函数，无需了解函数的内部的实现，
10      只需要掌握被调用函数的(三要素)：功能，参数，返回值
```

2.2 定义函数的语法格式

```
1 返回类型   函数名 (形参列表)
```

```
{
    函数体;
    return 返回值;
}

返回类型  函数名 (数据类型 形参变量名1, 数据类型 形参变量名2,...)
{
    函数体;
    return 返回值;
}

注:
    1. 返回类型 :  数据类型
    2. 函数名:遵循标识符的命名的规则
    3. 形参变量名 : 变量名,遵循标识符的命名的规则,
        此变量属于局部变量,只能在函数内被调用。

    4. 如果函数有返回值通过 return 返回值;进行返回
        如果函数没有返回值,return;可以省略不写,
        或者写成return;
    5. 函数的形参也是可有可无,如果没有形参写成()或者(void)
```

2.3 函数的调用

```
函数没有形参，没有返回值：
    函数名();        ---> ()中不可以写任何的东西

函数有形参，没有返回值：
    函数名(实参列表);   ---> 比如：函数名(1,2);

函数没有形参，有返回值：
    变量名 = 函数名();

函数有形参，有返回值：
    变量名 = 函数名(实参列表);   ---> 比如:sum = 函数名(a, b);

如果函数的返回类型为int类型，一般使用int类型的变量接收函数的返回值，
如果使用float类型的变量接收返回类型为int类型的函数的返回值，
会发生隐式类型转换。
```

2.4 函数的声明

```
    如果函数的定义被写到函数调用的后边，需要在调用函数之前，
    对被调用的函数进行声明。


    // 函数声明的格式：
```

```c
    // 返回类型  函数名(数据类型 形参名, 数据类型 形参名,...);
    // 或者
    // 返回类型  函数名(数据类型 , 数据类型 ,...);

    int add_func(int a, int b);  // 函数的声明
    // int add_func(int , int );  // 函数的声明
    int main(int argc, char *argv[])
    {
        // int add_func(int a, int b); // 函数声明, 只能在此函数中被调用
        printf("%d\n", add_func(100,200));
        return 0;
    }
    int add_func(int a, int b)
    {
        return a+b;
    }
```

2.5 函数的形参

2.5.1 普通类型的形参(值传递)

```
int add_func(int a, int b)          // 使用值传递即可
{
    return a+b;
}

int add_func(int *a, int *b)          // 使用地址传递
{
    return *a+*b;
}
```

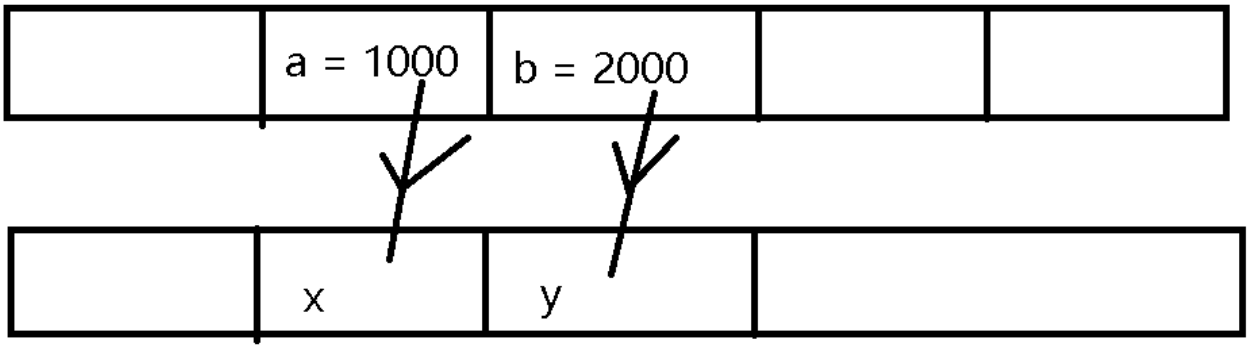
2.5.2 一级指针类型的形参(地址传递)

```
实现一个函数，通过函数交互两个变量的值
#include <stdio.h>
void show(int a, int b)
{
    printf("a = %d, b = %d\n", a, b);
}
void swap(int x, int y)
{
```

```c
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
}

void swap2(int *p, int *q)
{
    *p = *p ^ *q;
    *q = *p ^ *q;
    *p = *p ^ *q;

    // *p = 1000 1000
    // *q = 0111 1010
    // *p = *p ^ *q = 1111 0010
    // *q = *p ^ *q = 1000 1000
    // *p = *p ^ *q = 0111 1010
}

int main(int argc, const char *argv[])
{
    int a = 1000, b = 2000;
    // 编写函数实现a和b变量的值的交换
    printf("交换之前 >");
    show(a,b);
    swap(a, b);   // 值传递，不可以进行交换，
    printf("交换之后 >");
    show(a, b);

    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。

```
1 定义函数，通过参数返回函数的结果
2 #include <stdio.h>
```

```c
// 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;
}

int main(int argc, const char *argv[])
{
    int sum = add_func(100,200);
    printf("sum = %d\n", sum);

    int mul;
    mul_func(100,200,&mul);
    printf("mul = %d\n", mul);
    return 0;
}

```

2.5.3 二级指针类型的形参(地址传递)

2.6 一维数组作为函数的参数(一级指针)

2.7 二维数组作为函数的参数(数组指针)

2.8 指针数组作为函数参数(二级指针)

```
下周一授课内容:
main函数的参数
函数指针
指针函数
函数指针数组
函数指针数组指针
```