指针初阶

1. 指针是什么?

指针理解的2个要点:

```
1. 指针是内存中一个最小单元的编号，也就是地址
1. **平时口语中说的指针，通常指的是指针变量**，是用来存放内存地址的变量
```

总结：指针就是地址，口语中说的指针通常指的是指针变量

> 指针变量
>
> 我们可以通过&（取地址操作符）取出变量的内存其实地址，把地址可以存放到一个变量中，这个变量就是指针变量

存放地址的变量指针变量

```c
#include <stdio.h>
int main()
{
 int a = 10;//在内存中开辟一块空间
 int *p = &a;//这里我们对变量a，取出它的地址，可以使用&操作符。
    //a变量占用4个字节的空间，这里是将a的4个字节的第一个字节的地址存放在p变量
中，p就是一个指针变量。
 return 0;
}
```

总结:

指针变量，用来存放地址的变量。（存放在指针中的值都被当成地址处理）。

那这里的问题是:

- 一个小的单元到底是多大?
- （1个字节） 如何编址?

经过仔细的计算和权衡我们发现一个字节给一个对应的地址是比较合适的。

00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000001

...

11111111 11111111 11111111 11111111

这里就有2的32次方个地址。

每个地址标识一个字节

那我们就可以给 （2^32Byte == 2^32/1024KB == $2^{32/1024/1024MB==2}$32/1024/1024/1024GB == 4GB） 4G的空闲进行编址。

同样的方法，那64位机器，如果给64根地址线

这里我们就明白:

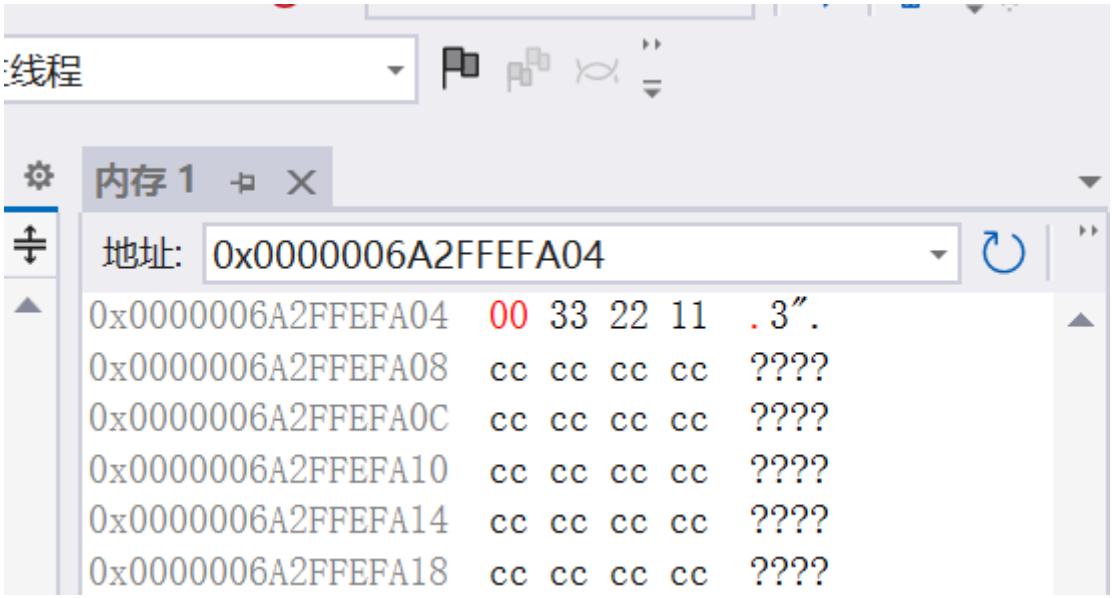
在32位的机器上，地址是32个0或者1组成二进制序列，那地址就得用4个字节的空间来存储，所以 一个指针变量的大小就应该是4个字节。 那如果在64位机器上，如果有64个地址线，那一个指针变量的大小是8个字节，才能存放一个地 址。

为什么地址要用四个字节来储存?

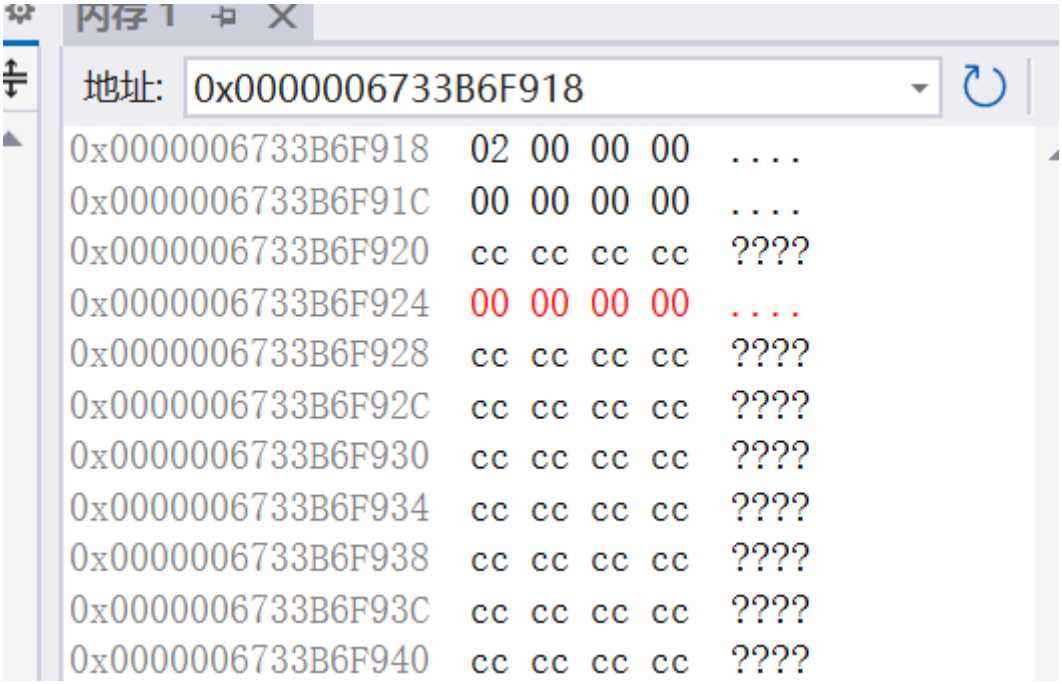
因为在32位机器上地址是由32个0或者1组成的二进制序列,一字节等于8个byte

- 每一个字节都有一个编号，这个编号就是地址，地址在C语言中称为**指针**
- 内存被分为一个个的内存单元，每个内存单元的大小是一个字节
- 地址要储存的话，存放在指针变量中
- 例如:int* p，int指向对象类型，*的话说明p的类型是指针,p是**指针变量**,用来储存地址
- 在32位机器上地址的大小是4个字节，所以**指针变量的大小是4个字节**
- 注意:并不是说每一个地址都要用指针变量储存下来，如果那样就太浪费空间了，我们只是储存我们可能用到的

```c
//当指针类型是char*类型时,解引用的时候访问一个字节
int main() {
    int a = 0x11223344;
    char* p = (char*) & a;
    *p = 0;//解引用操作符,间接访问
    //a = 0;
    return 0;
}
```




```c
//当指针类型时int*类型时,解引用时访问四个字节
int main() {
    int a = 0x11223344;
    int* p = &a;
    *p = 0;//解引用操作符,间接访问
    //a = 0;
    return 0;
}
```



指针类型是有意义的

- **指针类型决定了指针进行解引用操作的时候，访问几个字节**
- 指针类型决定了指针+1/-1跳过几个字节
  - int* 的指针+1，跳过四个字节
  - char* 的指针+1，跳过1个字节
  - short* 的指针+1，跳过2个字节
  - double* 的指针+1, 跳过8个字节

```c
int main() {
    int a = 0;
    int* pa = &a;
    char* pc = (char*) & a;
    printf("pa = %p\n", pa);
    printf("pa + 1 = %p\n", pa + 1);
    printf("pc + 1 = %x\n", pc + 1);
    printf("pc = %p\n", pc);
}
```

```
pa = 000000A629BAF884
pa + 1 = 000000A629BAF888
pc + 1 = 29baf885
pc = 000000A629BAF884
```

2. 指针和指针类型

这里我们在讨论一下：**指针的类型**

我们都知道，变量有不同的类型，整形，浮点型等。

那指针有没有类型呢?

准确的说：有的。

当有这样的代码：

```
int num = 10;
p = &num;
```

要将&num（num的地址）保存到p中，我们知道p就是一个指针变量

那它的类型是怎样的呢？ 我们给指针变量相应的类型。

```
char  *pc = NULL;
int   *pi = NULL;
short *ps = NULL;
long  *pl = NULL;
float *pf = NULL;
double *pd = NULL;

```

一般空指针设置为null，而不是0或者其他，像0,它本身便有歧义，比如: 它可能是字符0

简单来说: 当我们看见null时，**便知道它时空指针**

这里可以看到，指针的定义方式是： type + * 。

其实：

char* 类型的指针是为了存放 char 类型变量的地址。

short* 类型的指针是为了存放 short 类型变量的地址。

int* 类型的指针是为了存放 int 类型变量的地址。

那指针类型的意义是什么?

2.1 指针+-整数

```
#include <stdio.h>
//演示实例
int main()
{
 int n = 10;
 char *pc = (char*)&n;
 int *pi = &n;

 printf("%p\n", &n);
 printf("%p\n", pc);
 printf("%p\n", pc+1);
 printf("%p\n", pi);
 printf("%p\n", pi+1);
 return  0;
}
```

总结: **指针的类型决定了指针向前或者向后走一步有多大距离**

2.2 指针的解引用

```c
//演示实例
#include <stdio.h>
int main()
{
 int n = 0x11223344;
 char *pc = (char *)&n;
 int *pi = &n;
 *pc = 0;    //重点在调试的过程中观察内存的变化。
 *pi = 0;    //重点在调试的过程中观察内存的变化。
 return 0;
}

```

总结： 指针的类型决定了，对指针解引用的时候有多大的权限（**能操作几个字节**）。

比如： char* 的指针解引用就**只能访问一个字节**，而 int* 的指针的解引用就**能访问四个字节。**

3. 野指针

概念： **野指针就是指针指向的位置是不可知的**（随机的、不正确的、没有明确限制的）

3.1 野指针成因

1. **指针未初始化**

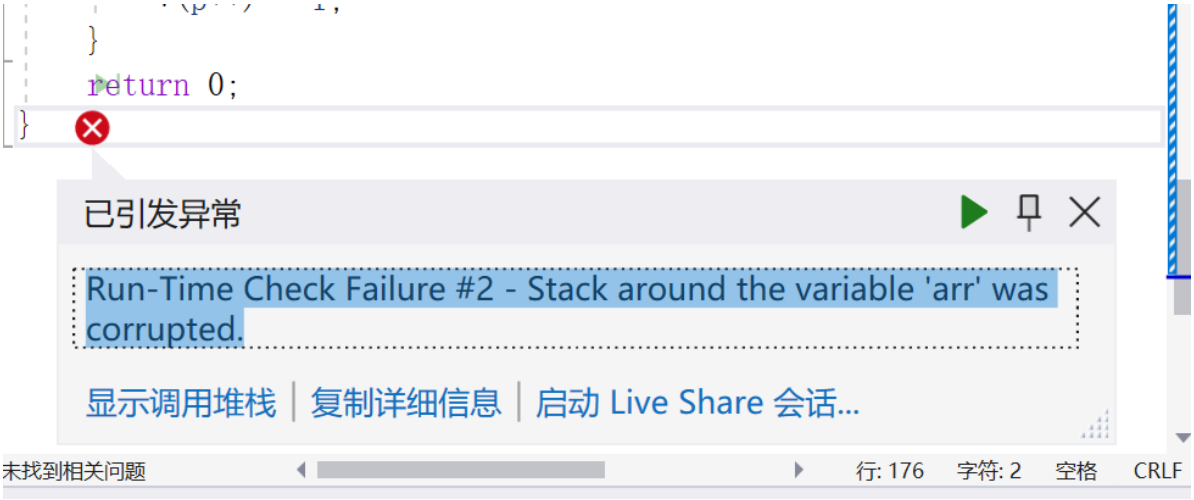
```c
#include <stdio.h>
int main()
{
 int *p;//局部变量指针未初始化，默认为随机值
    *p = 20;
 return 0;
}

```

2. **指针越界访问**

```c
#include <stdio.h>
int main()
{
    int arr[10] = {0};
    int *p = arr;
    int i = 0;
    for(i=0; i<=11; i++)
    {
        //当指针指向的范围超出数组arr的范围时，p就是野指针
        *(p++) = i;
    }
    return 0;
}

```



3. **指针指向的空间释放**

动态内存开辟

3.2 **如何规避野指针**

1. 指针初始化
2. 小心指针越界
3. **指针指向空间释放即使置**NULL
4. **避免返回局部变量的地址** (局部变量使用后便会被摧毁,地址也就不复存在了,也就出现了野指针)
5. 指针使用之前检查有效性

```c
#include <stdio.h>
int main()
{
    int *p = NULL;
    //....
    int a = 10;
    p = &a;
    if(p != NULL)
   {
        *p = 20;
   }
    return 0;
}
```

总结:

1. 如果明确指针应该指向哪里，就初始化正确的地址

```c
int* test()
{
    int a = 10;//0x0012ff40
    return &a;
}

int main()
{
    //0x0012ff40
    int *p = test();
    //p就是野指针
    printf("%d\n", *p);//
```

```
    return 0;
}
```

1. 如果指针不知道初始化什么值，为了安全，初始化NULL(`int* p = NULL`)

```
int main()
{
    int* p = NULL;
    if (p != NULL)
    {
        //..
    }

    return 0;
}
```

一般使用格式

4. 指针运算

- 指针+- 整数

```
#define N_VALUES 5
float values[N_VALUES];
float *vp;
//指针+-整数；指针的关系运算
for (vp = &values[0]; vp < &values[N_VALUES];)
{
     *vp++ = 0;
}
//正序
```

```
#define N_VALUES 5
float values[N_VALUES];
float *vp;
//指针+-整数；指针的关系运算
for (vp = &values[N_VALUES]; vp < &values[0];)
{
     *--vp = 0;
}
//倒叙,一般不行
```

实际在绝大部分的编译器上是可以顺利完成任务的，然而我们还是应该避免这样写，因为标准并不保证它可行。

标准规定:

允许指向数组元素的指针与指向数组**最后一个元素后面**的那个内存位置的指针比较，但是**不允许与**

指向第一个元素之前的那个内存位置的指针进项比较

- 指针-指针

返回的是指针与指针之间的元素个数

```c
int main()
{
    int arr[10] = { 0 };
    //
    //指针-指针的前提：两个指针指向同一块区域，指针类型时相同的
    //指针-指针差值的绝对值，指针和指针之间的元素个数
    //
    printf("%d\n", &arr[9] - &arr[0]);
    printf("%d\n", &arr[0] - &arr[9]);

    return 0;
}
```

```c
//求数组的长度
size_t my_strlen(char* str)
{
    char* start = str;
    while (*str)    //while(*str != \0)
    {
        str++;
    }
    return str - start;
}

int main() {
    char arr[] = "abcd";
    size_t len = my_strlen(arr);
    printf("%zu\n", len);
    return 0;
}
```

`size_t`类型通常用于以下情况：

1. 表示内存大小：`size_t`用于表示内存块的大小，例如在动态内存分配函数`malloc`、`calloc`、`realloc`中指定要分配的字节数。
2. 表示数组或字符串的长度：`size_t`**用于表示数组或字符串的长度**，例如在循环中迭代数组或字符串的元素。
3. 存储对象的大小：`size_t`用于表示对象的大小，例如在`sizeof`运算符中获取对象的大小。

在这些情况下，使用`size_t`类型可以提高代码的可移植性，因为它的大小会根据不同的编译器和操作系统进行适当的调整。此外，`size_t`是无符号整数类型，可以确保表示非负的大小或索引值。

需要注意的是，当涉及到负数或需要进行符号运算时，应该使用带符号的整数类型，如`ssize_t`。

5. 指针与数组

指针就是指针，指针就是地址，指针变量就是就是一个变量，用来存放地址，**指针变量的大小是**4/8

- 指针的关系运算

```c
int main()
```

```c
{
    int arr[] = { 1,2,3,4,5,6,7,8,9,10 };
    //            0 1 2 3 4 5 6 7 8 9
    //使用指针打印数组的内容
    int * p = arr;
    int i = 0;
    //arr-->p
    //arr == p
    //arr+i  ==  p+i
    //*(arr+i) == *(p+i) == arr[i]
    //*(arr+i) == arr[i]
    //*(i+arr) == i[arr]
    //3+5
    //5+3
    for (i = 0; i < 10; i++)
    {
        //printf("%d ", *(p + i));
        printf("%d ", *(arr + i));
        //printf("%d ", arr[i]);
        //printf("%d ", i[arr]);

        //p指向的是数组首元素
        //p+i 是数组中下标为i的元素的地址
        //p+i 起始时跳过了i*sizeof(int)个字节
    }
    return 0;
}
```

数组打印内容的实际意义***(arr+i) == *(p+i) == arr[i]**

arr[i]只是数组的表现形式，实际上是我们知道了某个数组的首地址，然后根据首地址来进行偏移来访问数组内容

当然也可以是i[arr]

数组首元素地址与数组地址的区别

- 数组的数组名是数组首元素的地址，地址是可以访问指针变量　通过指针可以访问一个数组的元素
- **数组名表示数组首元素的地址**

````
既然可以把数组名当成地址存放到一个指针中，我们使用指针来访问一个就成为可能

```c
#include <stdio.h>
int main()
{
    int arr[] = {1,2,3,4,5,6,7,8,9,0};
    int *p = arr; //指针存放数组首元素的地址
    int sz = sizeof(arr)/sizeof(arr[0]);
    for(i=0; i<sz; i++)
   {
        printf("&arr[%d] = %p   <=====> p+%d = %p\n", i, &arr[i], i, p+i);
   }
    return 0;
}
```
````

```

所以 p+i 其实计算的是数组 arr 下标为i的地址。

那我们就可以直接通过指针来访问数组。

```c
int main()
{
 int arr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
 int *p = arr; //指针存放数组首元素的地址
 int sz = sizeof(arr) / sizeof(arr[0]);
 int i = 0;
 for (i = 0; i<sz; i++)
 {
 printf("%d ", *(p + i));
 }
 return 0;
}
```

但是有两个例外:

1. sizeof(数组名)，数组名单独放在sizeof内部，数组名表示整个数组，单位是字节
2. &数组名，数组名表示整个数组，取出的是数组的地址
3. 数组的地址与数组首元素的地址，值是一样的，**但是类型和意义是不一样的**

示例:

```c
int main() {
    int arr[10] = { 0 };
    printf("%p\n", arr);//打印数组首地址
    printf("%p\n", arr + 1);//int*跳过四个字节

    printf("%p\n", &arr[0]);//打印数组首地址
    printf("%p\n", &arr[0] + 1);//跳过四个字节

    printf("%p\n", &arr);//取出数组的地址
    printf("%p\n", &arr + 1);//跳过整个数组的字节

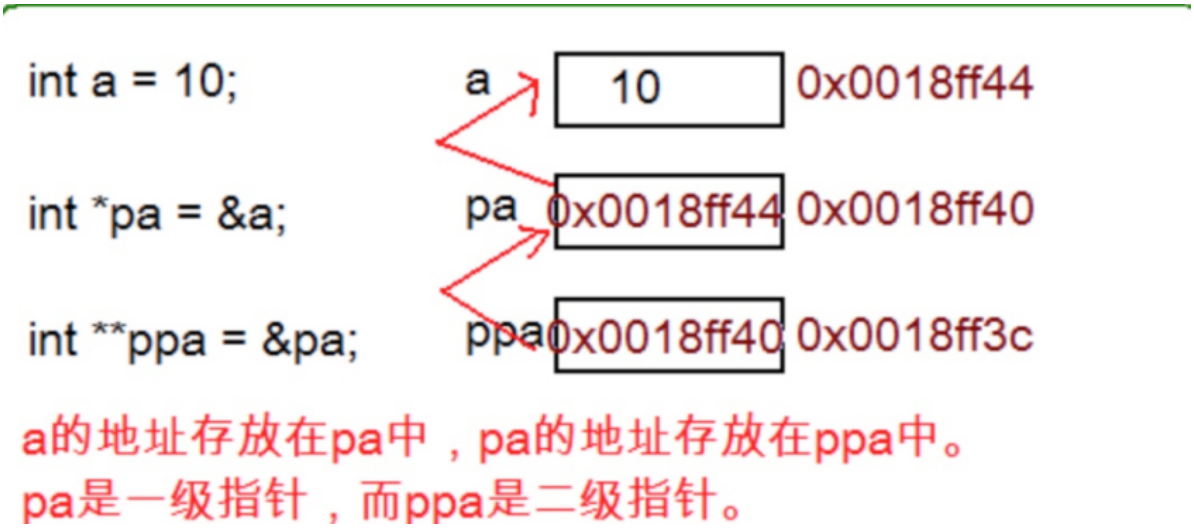
    printf("%d\n", sizeof(arr));
    return 0;
}
```
```

```
000000C40912FA78
000000C40912FA7C
000000C40912FA78
000000C40912FA7C
000000C40912FA78
000000C40912FAA0
40
```

6. 二级指针

指针变量也是变量，是变量就有地址，那指针变量的地址存放在哪里？ 这就是 二级指针

二级指针变量是存放一级指针变量的地址的



对于二级指针的运算有：

- ***ppa 通过对ppa中的地址进行解引用，这样找到的是 pa， *ppa 其实访问的就是 pa**

```
int b = 20;
*ppa = &b;//等价于 pa = &b;
```

- ****ppa 先通过 *ppa 找到 pa ,然后对 pa 进行解引用操作： *pa， 那找到的是 a**

```
**ppa = 30;
//等价于*pa = 30;
//等价于a = 30;
```

```c
int main()
{
    int a = 10;
    int* p =  &a;//p是指针变量，一级指针变量
    int* * pp = &p;//pp指针变量，二级指针变量

    **pp = 20;
    printf("%d\n", a);//20

    //int** * ppp = &pp;//pp是指针变量，三级指针变量
    //...
    return 0;
}
```

7. 指针数组

指针数组是指针还是数组?

答案：是数组。

是存放指针的数组。

数组我们已经知道整形数组，字符数组。

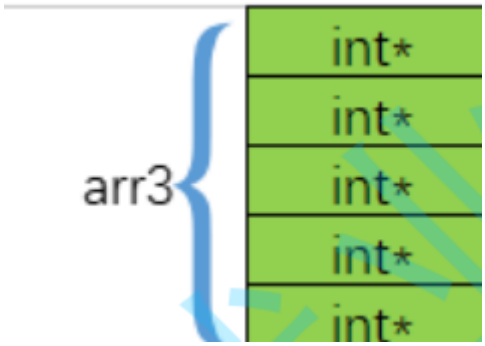
```c
int arr1[5];
char arr2[6];
```

那指针数组是怎样的?

```c
int* arr3[5];//是什么?

```

arr3是一个数组，有五个元素，每个元素是一个整形指针。



字符数组 - 存放字符的数组
整型数组 - 存放整型的数组
指针数组 - 存放指针（地址）的数组

```
char arr[7];
int arr[6];
```

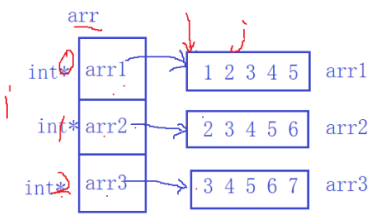
```
char*
double*
int*
```

```
char* arr[5];//存放字符指针的数组
double* arr2[4];//存放字符指针的数组
```

```
int main()
{
    //使用指针数组，模拟一个二维数组
    int arr1[] = { 1,2,3,4,5 };
    int arr2[] = { 2,3,4,5,6 };
    int arr3[] = { 3,4,5,6,7 };

    //指针数组
    int* arr[] = { arr1, arr2, arr3 };

    return 0;
}
```




```
int main()
{
    //使用指针数组，模拟一个二维数组
    int arr1[] = { 1,2,3,4,5 };
    int arr2[] = { 2,3,4,5,6 };
    int arr3[] = { 3,4,5,6,7 };

    //指针数组
    int* arr[] = { arr1, arr2, arr3 };

    int i = 0;
    for (i = 0; i < 3; i++)
    {
        int j = 0;
        for (j = 0; j < 5; j++)
        {
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```


1 2 3 4 5
2 3 4 5 6
3 4 5 6 7