第八章 程序设计复合类型-指针

模块8.1：指针基础

主讲教师：同济大学计算机科学与技术学院 陈宇飞
同济大学计算机科学与技术学院 龚晓亮

目录

- 指针的基本概念
- 指针的声明与初始化
- 指针的使用
- 指针作函数的参数
- 指针的危险

目录

- 指针的基本概念
  - 引言
  - 指针概念
  - 变量与指针
  - 指针和数字

1.1 引言

✓计算机程序存储数据必须要跟踪的3个基本属性：

- ❖ 信息存储在何处（地址）
- ❖ 存储的值为多少（值）
- ❖ 存储的信息是什么类型（空间大小）

✓另一种策略以指针为基础，指针是一个变量，其存储的值是地址

✓查找常规变量的地址，使用地址运算符（&）

1.1 引言

✓面向对象编程（OOP）强调在运行阶段（而不是编译阶段）进行决策

✓运行阶段决策可以根据程序正在运行当时的情况进行调整

✓例如：数组长度在程序编译时就设定好，这是编译阶段决策

✓OOP可以在运行阶段确定数组的长度，使用关键字new请求正确数量的内存以及使用指针来跟踪新分配的内存的位置（后续内容）

1.1 引言

- ✓指针是高级编程语言中非常重要的概念，它能使得不同区域的代码可以轻易的共享内存数据（指针能使一个函数访问另一个函数的局部变量，指针是两个函数进行数据交换必不可少的工具）
- ✓指针使得构建复杂链接性的数据结构成为可能，有些操作必须使用指针，比如申请堆内存，还有C++或者C语言中函数的调用，如果在函数中修改被传递的对象，就必须通过对象指针来完成

1.1 引言

✓函数的缺陷（函数返回中return语句的局限性）

- ❖一个函数只能返回一个值，就算我们在函数里面写多个return语句，但是只要执行任何一条return语句，整个函数调用就结束了
- ❖数组可以帮助我们返回多个值，但是数组是相同数据类型的结合，对于不同数据类型则不能使用数组

使用指针可以有效解决这个问题

1.1 引言

✓数据在内存中的存放

- ❖ 根据不同的类型存放在动态/静态数据区
- ❖ 数据所占内存大小由变量类型决定 sizeof(类型)

✓内存地址

- ❖ 给内存中每一个字节的编号
- ❖ 内存地址的表示根据内存地址的大小一般分为16位、32位和64位(一般称为地址总线的宽度，是CPU的理论最大寻址范围，具体还受限于其它软、硬件)

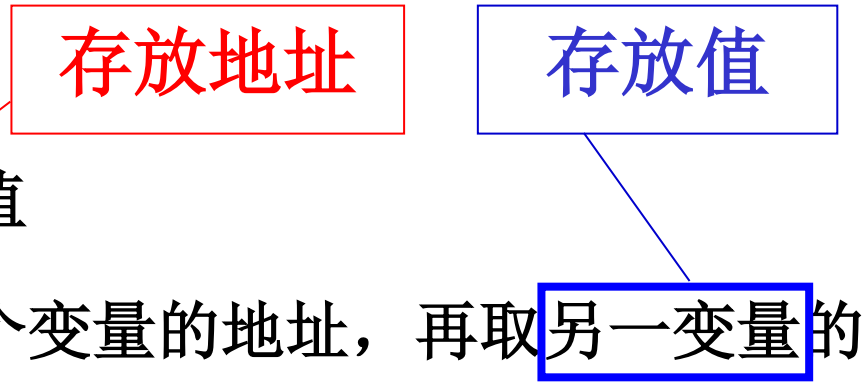
1.1 引言

✓内存中的内容

- ❖ 以字节为单位，用一个或几个字节来表示某个数据的值(基本数据类型一般都是2的n次方)

✓内存中内容的访问

- ❖ 直接访问：按变量的地址取变量值
- ❖ 间接访问：通过某个变量取另一个变量的地址，再取另一变量的值



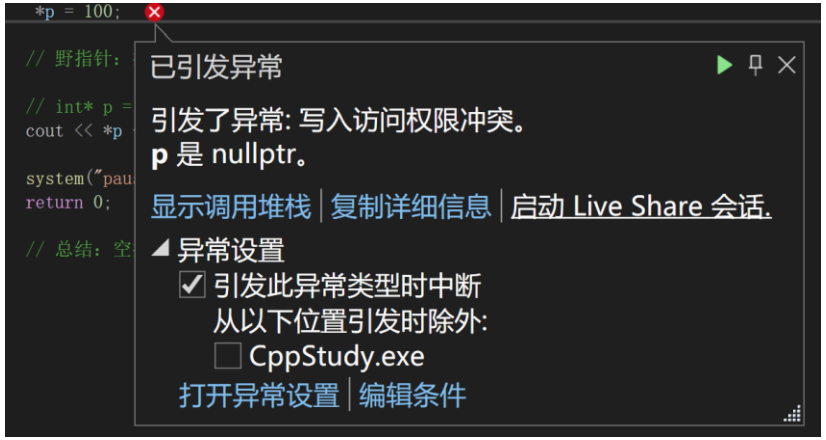
1.2 指针概念

✓特殊类型的变量-指针，用于存储值的地址

✓指针名表示的是地址

✓*运算符被称为间接值（indirect value）或解除引用（dereferencing），将其应用于指针，可以得到该地址处存储的值（C++根据上下文来自动确定所指的是乘法还是解除引用）

1.2 指针概念



✓空指针

❖空指针就是指针变量指向的内存编号为0的空间，我们可以使用空指针来初始化指针变量

❖这里需要注意的是，空指针指向的内存是不可以被访问的

```
int *p = NULL;
*p = 100; // 错误
```

❖因为空指针指向的内存是不可以访问的,0~255之间的内存是系统占用的，不可以访问，访问的话就会报异常

1.2 指针概念

```
#include<iostream>
using namespace std;
int main() {

    //指针变量p指向内存地址编号为0的空间
    int* p = NULL;

    //访问空指针报错
    //内存编号0 ~255为系统占用内存，不允许用户访问
    cout << *p << endl;

    return 0;

}
```

1.2 指针概念

✓野指针

- 野指针就是指针变量指向非法的内存空间
- 需要注意的是，野指针指向的内存是不可以被访问的

```
int* p = (int*)0x1100
```

- 0x1100这个地址不知道是谁的，这种情况特别危险，如果这块地址有重要数据，随便分配并修改，就会导致重要数据丢失，所以野指针需要被避免

1.2 指针概念


```
#include<iostream>
using namespace std;
int main() {

    //指针变量p指向内存地址编号为0x1100的空间
    int* p = (int*)0x1100;

    //访问野指针报错
    cout << *p << endl;

    return 0;

}
```

尽量去访问自己申请的空间，空指针和野指针都不是我们申请的空间，因此不要访问

1.2 指针概念

✓指针所占空间

❖根据操作系统的位数不同，指针占的空间也不一样

❑在32位操作系统上，指针占4个字节

❑在64位操作系统上，指针占8个字节

❑所有的类型都一样，不管是整型指针，还是浮点型指针，都是一样的

1.2 指针概念

```
size of (int*)4
size of (float*)4
size of (double*)4
size of (char*)4
```

```
#include<iostream>
using namespace std;

int main() {
	// 指针所占内存空间，32位操作系统下，占用4个字节。64位下占用8个字节
	cout << "size of (int*)" << sizeof(int*) << endl;
	cout << "size of (float*)" << sizeof(float*) << endl;
	cout << "size of (double*)" << sizeof(double*) << endl;
	cout << "size of (char*)" << sizeof(char*) << endl;

	return 0;
}
```

1.3 变量与指针

✓指针的基本语法

```
int number = 10;
int *ptr; // 声明一个指向整数的指针
ptr = &number; // 将指针指向变量number的地址
```

✓当需要访问指针所指向的变量时，可以使用解引用操作符*

```
std::cout << *ptr; // 输出指针所指向的变量的值，这将输出 10
```

✓在C++中，要定义一个指针变量，需要使用星号（*）符号来表示该变量是一个指针

```
int *ptr;  // 定义一个指向整数的指针变量
double *dblPtr;  // 定义一个指向双精度浮点数的指针变量
char *charPtr;  // 定义一个指向字符的指针变量
```

1.3 变量与指针

✓可以声明多个指针变量

```
int *ptr1, *ptr2;  // 声明两个整数指针变量
```

✓为了使指针指向特定变量或内存地址，需要将其初始化为相应变量的地址

```
int num = 10;
int *ptr = &num;  // 将指针ptr初始化为变量num的地址
```

```cpp
#include<iostream>
using namespace std;

int main() {
	int a = 10;
	// 定义指针语法：数据类型 *指针变量名;
	int* p;
	// 1.让指针记录变量a的地址
	p = &a;
	// 2.使用指针
	cout << "a的值未被修改前：" << a << endl;
	*p = 100;
	cout << "a的地址：" << (int)&a << endl;
	cout << "指针p为：" << p << endl;
	cout << "指针指向的值为：a= " << a << " ,*p = " << *p << endl;

	return 0;
}
```

```
a的值未被修改前： 10
a的地址： 1238366260
指针p为： 0000003C49CFF834
指针指向的值为： a= 100 ,*p = 100
请按任意键继续. . .
```

1.4 指针和数字

✓指针不是整型，指针与整数是截然不同的类型

✓指针描述的是位置

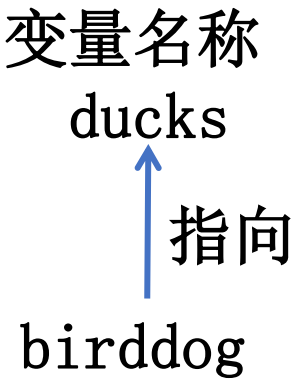
```
int * pt;
pt = 0xB8000000; // type mismatch
pt = (int *) 0xB8000000; // types now match
```

目录

- 指针的声明与初始化

2.1 指针的声明与初始化

内存地址	值
1000	12
1002	
1004	
1006	1000
1008	
1010	
1012	
1014	
1016	




```
int ducks = 12;
```

创建ducks变量，将值12存储在该变量中

```
int *birddog = &ducks;
```

创建birddog变量，将ducks的地址存储在该变量中

指针存储地址

2.1 指针的声明与初始化

✓指针变量不仅仅是指针，而且是指向特定类型的指针

✓和数组一样，指针都是基于其他类型的

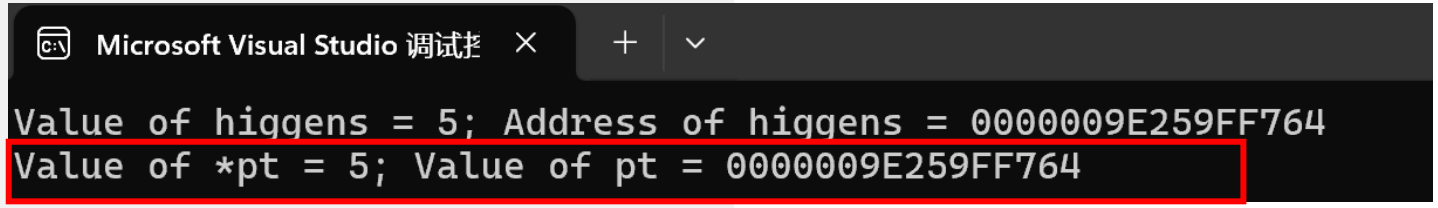
```
int * ptr; // Pointer to an integer
double * ptr; // Pointer to a double
char * ptr; // Pointer to a character
int * p1, p2; // p1 is a pointer to an integer, p2 is an integer
```

✓可以在声明语句中初始化指针。此时，被初始化的是指针，而不是它所指向的值

```
int higgens = 5;
int * pt = &higgens;
```

2.1 指针的声明与初始化

```
// init_ptr.cpp -- initialize a pointer
#include <iostream>
int main()
{
    using namespace std;
    int higgens = 5;
    int* pt = &higgens;

    cout << "Value of higgens = " << higgens
         << "; Address of higgens = " << &higgens << endl;
    cout << "Value of *pt = " << *pt
         << "; Value of pt = " << pt << endl;

    return 0;
}
```



2.1 指针的声明与初始化

✓由于指针数据的特殊性，他的初始化和赋值运算是有约束条件的，只能使用以下四种值:

(1)0值常量表达式

```
int *p1 = null;  //指针允许0值常量表达式
p1 = 0; //指针允许0只常量表达式
```

```
int a, z = 0;
int* p1 = a; //错误，地址初值不能是变量
int* p1 = z; //错误，整形变量不能作为指针，即使值为0
p1 = 4000; //错误，指针允许0值常量表达式
```

2.1 指针的声明与初始化

(2)相同指向类型的对象的地址

```
int a, * p1;
double f, * p3;
p1 = &a;
p3 = &f;
```

```
p1 = &f; //错误 p1和f指向类型不同
```

2.1 指针的声明与初始化

(3)相同指向类型的另一个有效指针

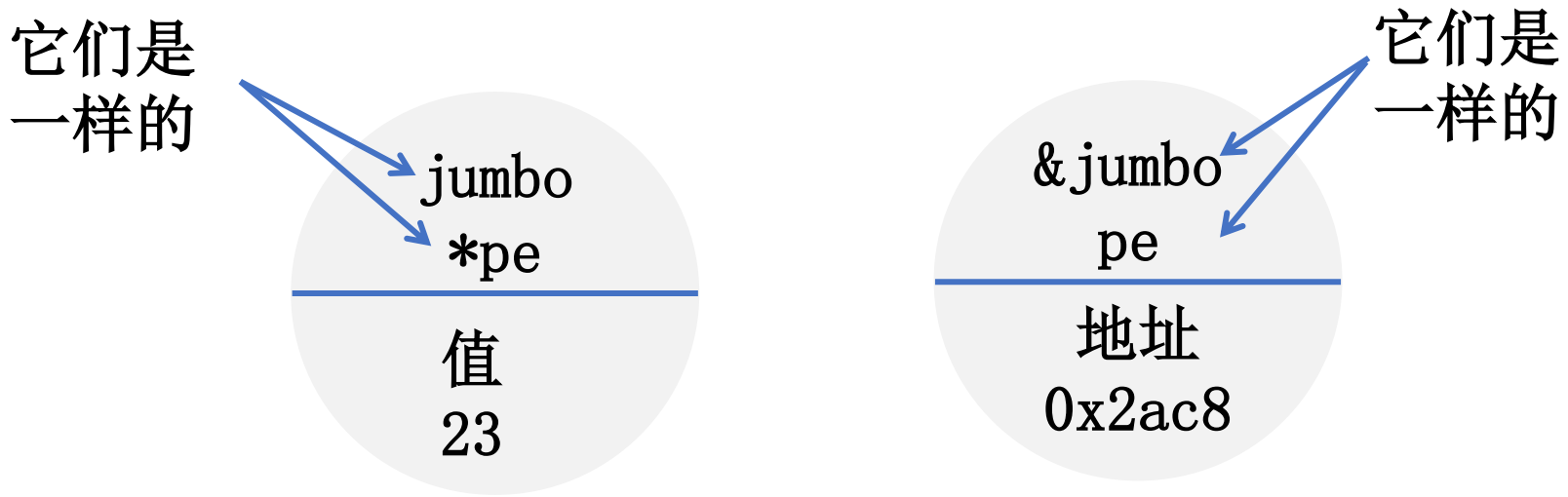
```
int x, * px = &x;
int* py = px; //相同指向类型的另一个指针
```

(4)对象存储空间后面下一个有效地址，如数组下一个元素的地址

```
int a[10], * px = &a[2];
int* py = &a[++i];
```

2.1 指针的声明与初始化

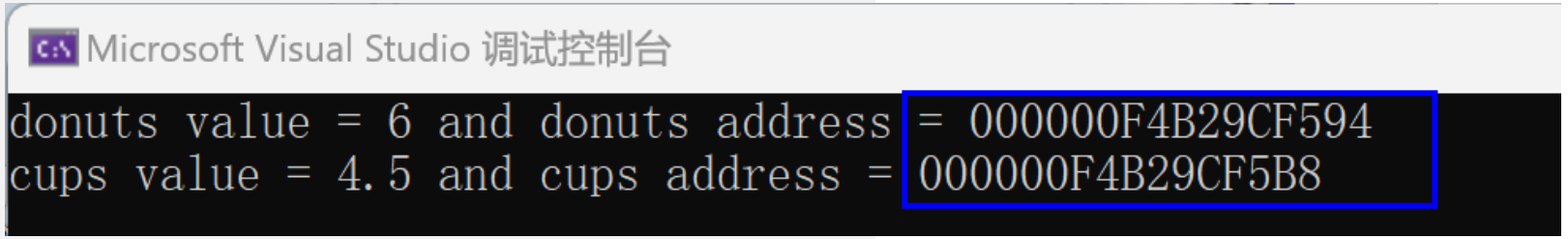
```
int jumbo = 23;
int * pe = &jumbo;
```



硬币的两面

2.1 指针的声明与初始化

```
// address.cpp -- using the & operator to find addresses
#include <iostream>
int main()
{
    using namespace std;
    int donuts = 6;
    double cups = 4.5;

    cout << "donuts value = " << donuts;
    cout << " and donuts address = " << &donuts << endl;
    // NOTE: you may need to use unsigned (&donuts)
    // and unsigned (&cups)
    cout << "cups value = " << cups;
    cout << " and cups address = " << &cups << endl;

    return 0;
}
```



使用常规变量时，值是指定的量，而地址为派生量

```
// pointer.cpp -- our first pointer variable
#include <iostream>
int main()
{
    using namespace std;
    int updates = 6;          // declare a variable
    int* p_updates;           // declare pointer to an int
    p_updates = &updates;     // assign address of int to pointer
    // express values two ways
    cout << "Values: updates = " << updates;
    cout << ", *p_updates = " << *p_updates << endl;
    // express address two ways
    cout << "Addresses: &updates = " << &updates ","<< endl;
    cout << " p_updates = " << p_updates << endl;
    // use pointer to change value
    *p_updates = *p_updates + 1;
    cout << "Now updates = " << updates << endl;

    return 0;
}
```

```
Values: updates = 6, *p_updates = 6
Addresses: &updates = 00000053FE98F974,
 p_updates = 00000053FE98F974
Now updates = 7
```

- 使用指针为变量时，地址是指定的量，而值为派生量
- 变量updates表示值，使用&运算符来获得地址
- 变量p_updates表示地址，使用*运算符来获得值
- 由于p_updates指向updates，因此*p_updates等价于updates

目录

- 指针的使用
  - 指针使用方法
  - const修饰指针

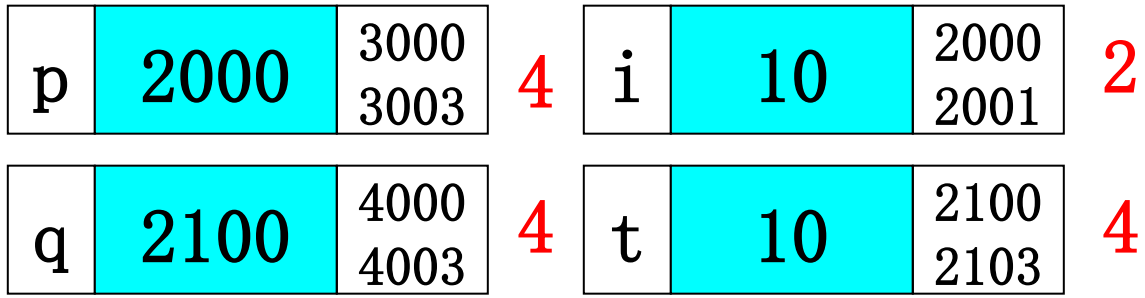
3.1 指针使用方法

- ✓ 指针变量名 ⟷ 地址
- ✓ *指针变量名 ⟷ 值

```
short i, *p;
p=&i;
*p=10; ⟷ i=10
```

```
long t, *q;
q=&t;
*q=10; ⟷ t=10
```

*p=10是间接访问
i=10 是直接访问



3.1 指针使用方法

✓ &与*的使用

- ❖ &表示取变量的地址，*表示取指针变量的值
- ❖ 两者优先级相同，右结合

```
int i=5, *p=&i;
&*p ⟷ &i ⟷ p
*&i ⟷ i
```

p	2000	3000 3003

i	5	2000 2003

变量定义时赋初值

```
int i=5, *p=&i;
```

用赋值语句赋值

```
int i=5, *p;
p=&i;
```

定义 *p;
赋值 p=&i;

3.1 指针使用方法

✓ 指针运算：指针运算都是作用在连续存储空间上才有意义

（1）指针加减整数运算

```
int x[10], n = 3, * p = &x[5];
p + 1          //指向内存空间中x[5]后面的第1个int型存储单元
p + n          //-------------------------n(3)个
p - 1          //--------------------前面----1个
p - n          //---------------------------n(3)个
```

3.1 指针使用方法

✓ 指针运算：指针运算都是作用在连续存储空间上才有意义

（2）指针变量自增自减运算

```
int x[10], * p = &x[5];
p++          //p指向x[5]后面的第1个int型内存单元
++p          //-----------------------------
p--          //p指向x[5]前面的第1个int型内存单元
--p          //-----------------------------
```

3.1 指针使用方法

（2）指针变量自增自减运算

- 指针变量的++/--单位是该指针变量的基类型
- void可以声明指针类型，但不能++/--

（void不能声明变量，但可以是函数的形参及返回值）

```
void  k;      × 错误，因为不知道该给k分配几字节的空间
void *p;      √ 正确，因为知道p大小是4字节
p++;          × 错误，因为不知道基类型的大小
p--;          × 错误，同上
```

3.1 指针使用方法

（2）指针变量自增自减运算

- *与++/--的优先级关系

*比后缀++/--优先级低　　*:3 后缀:2
*与前缀++/--优先级相同，右结合　　*:3 前缀:3

```
int i, *p=&i;
*p++  ⟷ *(p++)：保留p的旧值到临时变量中，p再++(不指向i)，
         最后取旧值所指的值(i)
*++p  ⟷ *(++p)：p先++(不指向i)，再取p的值(非i)
(*p)++ ⟷ i++ ：取p所指的值(i)，i值再后缀++
++*p  ⟷ ++i ：取p所指的值(i)，i值再前缀++
```

3.1 指针使用方法

（3）两个指针相减运算

设p1，p2是相同类型的两个指针，则p2 - p1的结果是两指针之间对象的个数，如果p2指针地址大于p1则结果为正，否则为负

```
int x[5], * p1 = &x[0], * p2 = &x[4];
int n;
n = p2 - p1; //n的值为4，即为他们之间间隔的元素的个数
```

运算方法：(p2储存的地址编码-p1储存的地址编码)/4;

若是double类型则除以8；char类型除以1

3.1 指针使用方法

（4）指针的运算关系

设p1、p2是同一个指向类型的两个指针，则p1和p2可以进行关系运算

```
int x[4], * p1 = &x[0], * p2 = &x[4];
p2 > p1;    //表达式为真
```

用于：比较这两个地址的位置关系，即哪一个是靠前或者靠后的元素

3.1 指针使用方法

```
#include <iostream>
using namespace std;
int main()
{
    short  s = 1, * p2 = &s;
    int  d = 0, * p5 = &d;

    cout << p2 << endl;
    cout << ++p2 << endl;
    cout << p5 << endl;
    cout << ++p5 << endl;
    return 0;
}
```

假设地址A

=地址A+2　2字节

假设地址B

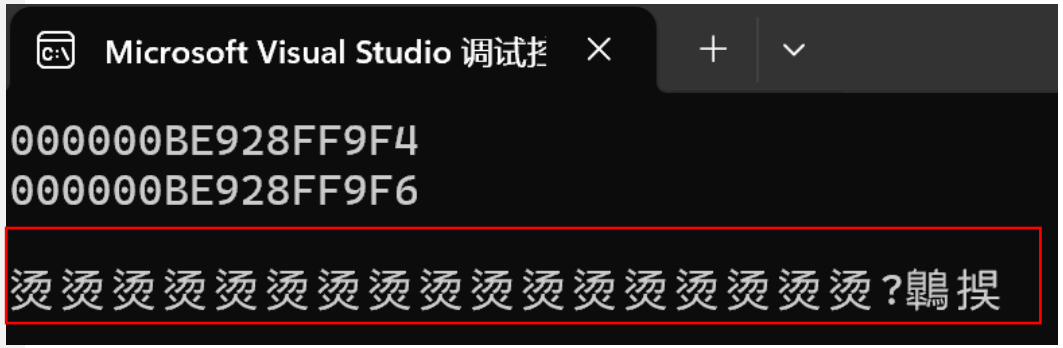
=地址B+4　4字节

3.1 指针使用方法

对于基类型为字符型的指针变量，
直接输出则不是输出地址，
而是表示以字符串方式输出
（依次输出到第一个\0为止）

```
#include <iostream>
using namespace std;
int main()
{
    short  s = 1, * p2 = &s;
    char d = 0, * p5 = &d;

    cout << p2 << endl;
    cout << ++p2 << endl;
    cout << p5 << endl;
    cout << ++p5 << endl;
    return 0;
}
```

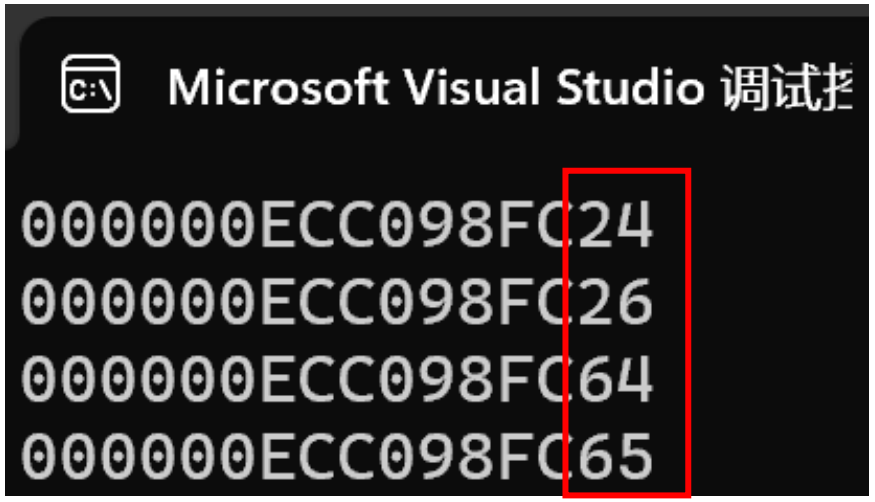
假设地址A
=地址A+2　2字节
一串乱字符
一串乱字符



3.1 指针使用方法

想输出字符型变量的地址，
先转为非char型才能输出

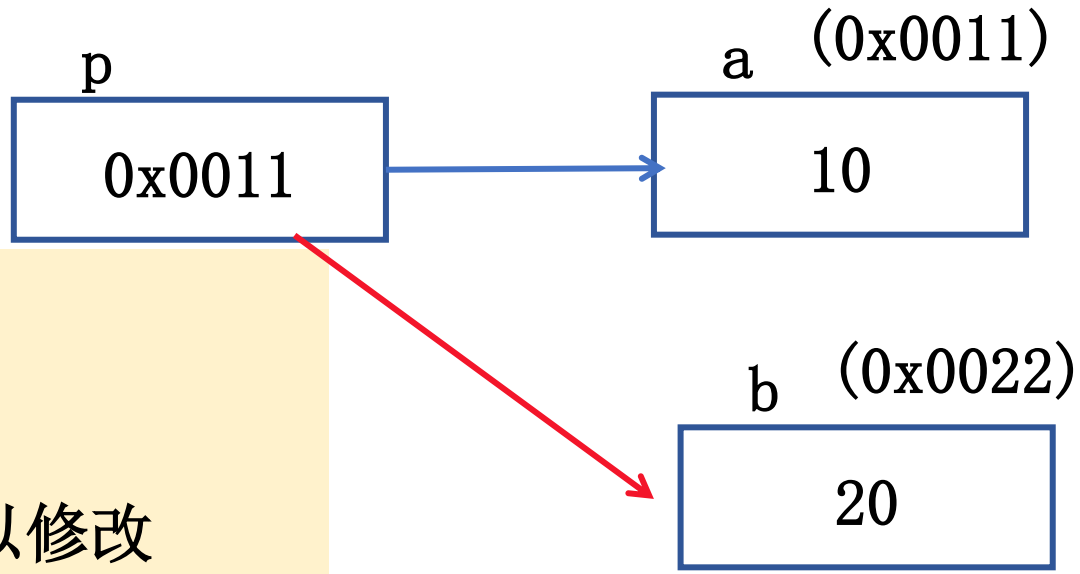
```
#include <iostream>
using namespace std;
int main()
{
    short  s = 1, * p2 = &s;
    char d = 0, * p5 = &d;

    cout << p2 << endl;                    地址A
    cout << ++p2 << endl;                  地址A+2
    cout << hex << (int*)(p5) << endl;     地址B
    cout << hex << (int*)(++p5) << endl;   地址B+1
    return 0;
}
```



3.2 const修饰指针

✓ const修饰指针有三种情况

（1）const修饰指针-常量指针




```
int a = 10;
int b = 20;
const int *p = &a;
*p = 20; //错误：指针指向的值不可以修改
p = &b; //正确：指针的指向可以修改
```

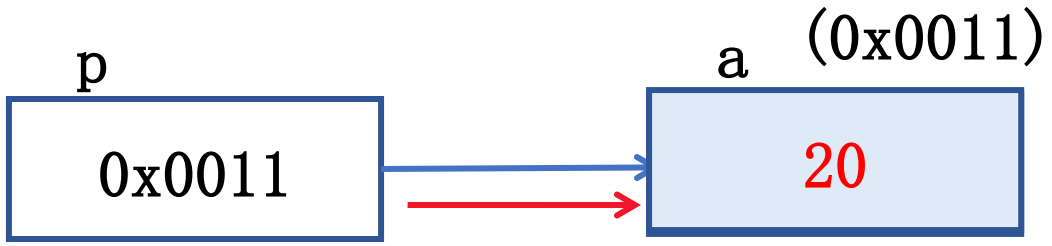
特点：

指针的指向可以修改，但是指针指向的值不可以修改

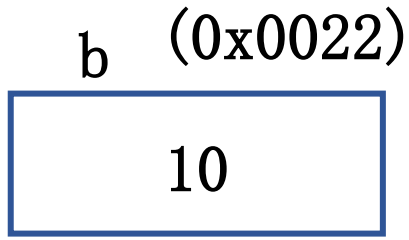
3.2 const修饰指针

✓ const修饰指针有三种情况

（2）const修饰常量-指针常量




```
int a = 10;
int b = 10;
int * const p = &a;
*p = 20; //正确：指针指向的值可以修改
p = &b; //错误：指针的指向不可以修改
```

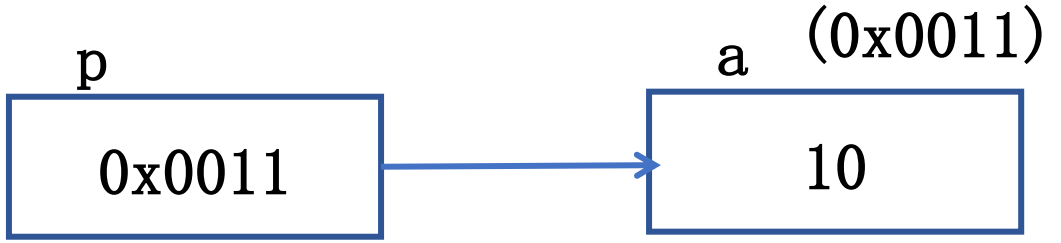


特点：指针的指向不可以改，指针指向的值可以修改

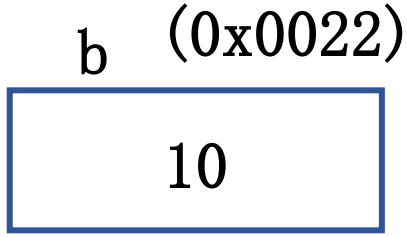
3.2 const修饰指针

✓ const修饰指针有三种情况

（3）const即修饰指针，又修饰常量




```
int a = 10;
int b = 10;
const int * const p = &a
*p = 20; //错误：指针指向的值不可以修改
p = &b; //错误：指针的指向不可以修改
```



特点：指针的指向和指向的值都不可以修改

技巧：看const右侧紧跟着的是指针还是常量，是指针就是常量指针，是常量就是指针常量

```
#include<iostream>
using namespace std;
int main() {
    int a = 10;
    int b = 10;
    //const修饰的是指针，指针指向可以改，指针指向的值不可以更改
    const int* p1 = &a;
    p1 = &b;            正确
    *p1 = 100;          报错

    //const修饰的是常量，指针指向不可以改，指针指向的值可以更改
    int* const p2 = &a;
    p2 = &b;            报错
    *p2 = 100;          正确

    //const既修饰指针又修饰常量
    const int* const p3 = &a;
    p3 = &b;            报错
    *p3 = 100;          报错

    return 0;
}
```

目录

- 指针作函数的参数

4.1 指针作函数的参数

✓指针变量做函数参数，虽然可以通过形参(实参地址)来间接访问实参，从而达到改变实参值的目的，但本质上仍然是单向传值（不是形参值回传实参）

✓指针变量做参数，可以同时得到多个改变的实参值，从而达到一个函数返回多个值的目的

✓必须通过改变形参指针变量所指变量(即实参)值的方法来达到改变实参值的目的，仅通过改变形参指针变量的值的方法是无效的

作用：利用指针作函数参数，可以修改实参的值

```cpp
#include<iostream>
using namespace std;
//值传递
void swap1(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}

//地址传递
void swap2(int* p1, int* p2)
{
    int temp = *p1;
    *p1 = *p2;
    *p2 = temp;

}
```

```cpp
int main() {
    int a = 10;
    int b = 20;
    swap1(a, b); // 值传递不会改变实参
    swap2(&a, &b); //地址传递会改变实参
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;

    return 0;
}
```

```
a = 20
b = 10
```

总结：如果不想修改实参，就用值传递，如果想修改实参，就用地址传递

Part1

```
#include<iostream>
using namespace std;
//冒泡排序函数
void bubbleSort(int* arr, int len)
//int * arr 也可以写为int arr[]
{
    for (int i = 0; i < len - 1; i++)
    {
        for (int j = 0; j < len - 1 - i; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

```
1
2
3
4
5
6
7
8
9
10
```

Part2

```
//打印数组函数
void printArray(int arr[], int len)
{
    for (int i = 0; i < len; i++)
    {
        cout << arr[i] << endl;
    }
}
```

Part3

```
int main() {
    int arr[10] = { 4,3,6,9,1,2,10,8,7,5 };
    int len = sizeof(arr) / sizeof(int);
    bubbleSort(arr, len);
    printArray(arr, len);

    return 0;
}
```

目录

- 指针的危险

5.1 指针的危险

✓在C++中创建指针时，计算机将分配用来存储地址的内存，但不会分配用来存储指针所指向的数据的内存

✓为数据提供空间是一个独立的步骤，不能忽略

```
long * fellow; // create a pointer-to-long
*fellow = 223323; // ✕ place a value in never-never land
```

✓一定要在对指针应用解除引用运算符（*）之前，将指针初始化为一个确定的、适当的地址

总结

- 指针的基本概念
  - 引言
  - 指针概念
  - 变量与指针
  - 指针和数字
- 指针的声明与初始化
- 指针的使用
  - 指针使用方法
  - const修饰指针
- 指针作函数的参数
- 指针的危险