复合类型：指针

COMP250205：计算机程序设计

李昊

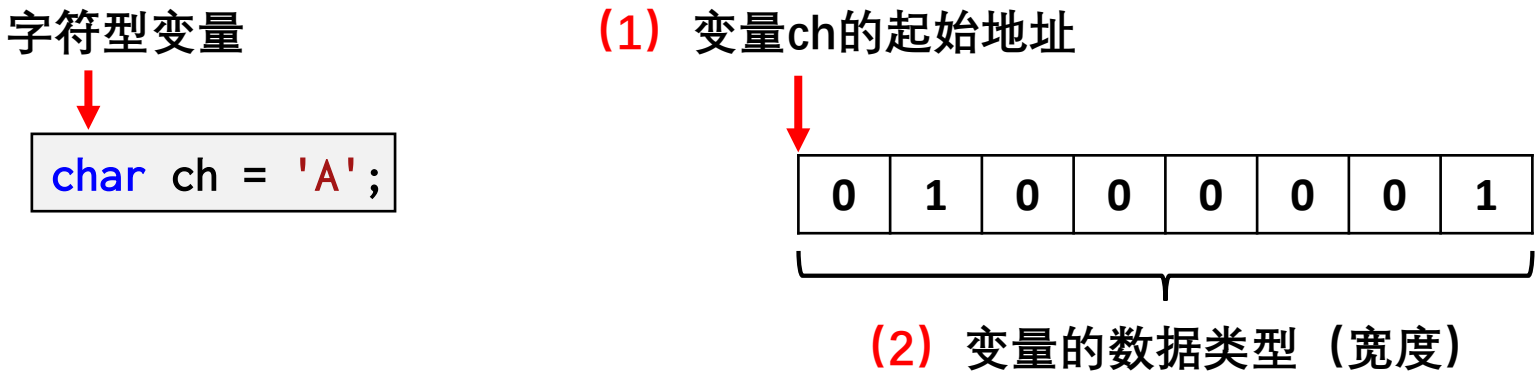
hao.li@xjtu.edu.cn

西安交通大学计算机学院

指针

重新考虑数据存储

存储和访问数据时的必要信息



换言之，通过这两个信息可以随意存储/访问数据

取地址：获取变量的地址

解引用：获取地址所对应的值

取地址与解引用

取地址操作符（&）获得变量的地址

解引用操作符（*）来获得对应地址里的值

```
#include <iostream>
int main() {
    using namespace std;
    int donuts = 6;

    cout << "donust value = " << donuts;
    cout << " and its address is " << &donuts;
    cout << " and its value from de-ref is " << *&donuts;
    cout << endl;

    return 0;
}
```

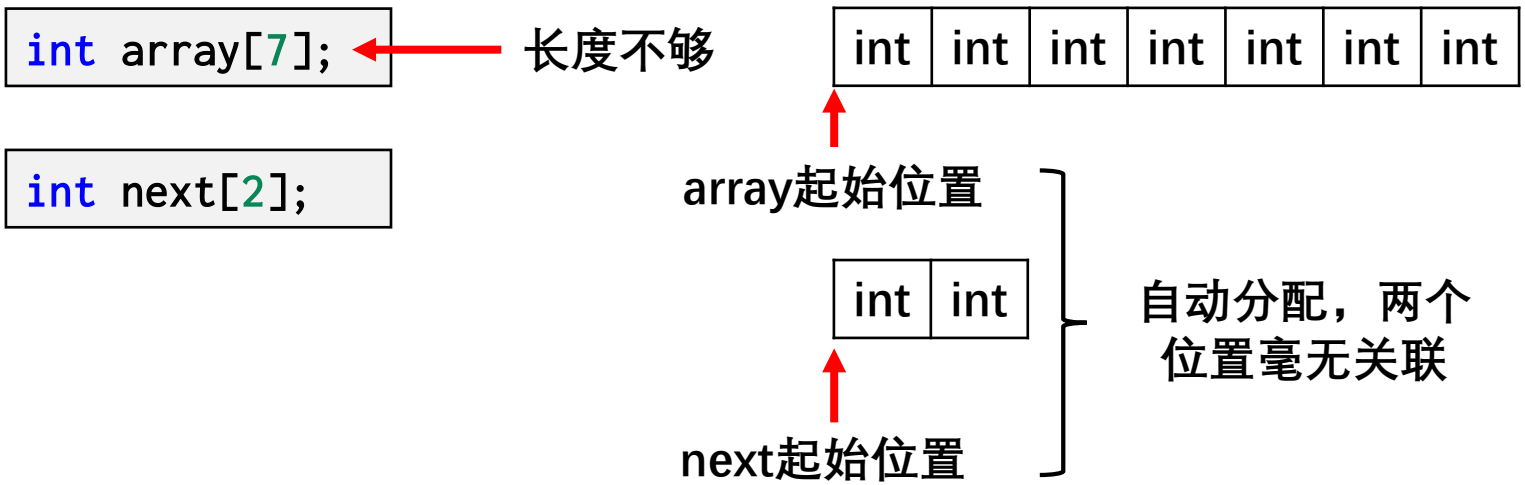
使用变量名访问

获取变量地址

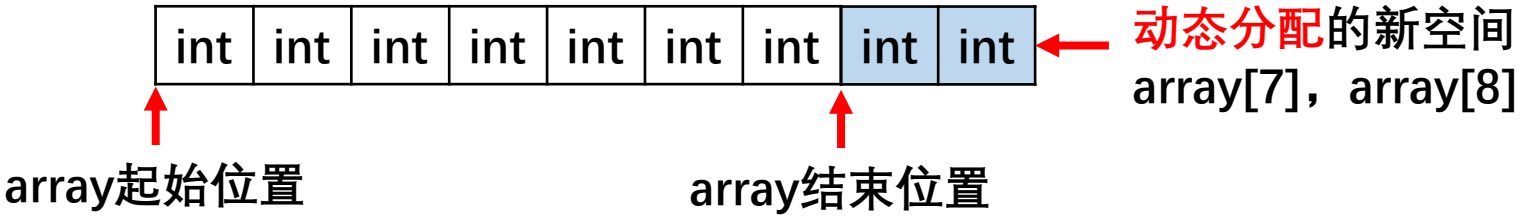
获取地址里的值

基于变量访问 vs. 基于地址访问 - 1

基于变量名的访问方式无法控制底层存储逻辑



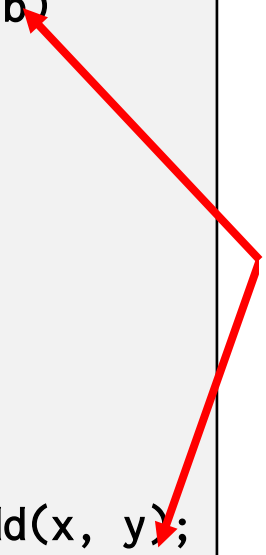
基于地址的访问方式可以对底层存储直接操作



基于变量访问 vs. 基于地址访问 - 2

函数的参数传递需要拷贝值

```
int add(int a, int b)
{
    return a + b;
}

int main()
{
    int x, y;
    cin >> x >> y;
    int result = add(x, y);
    cout << result << endl;
    return 0;
}
```



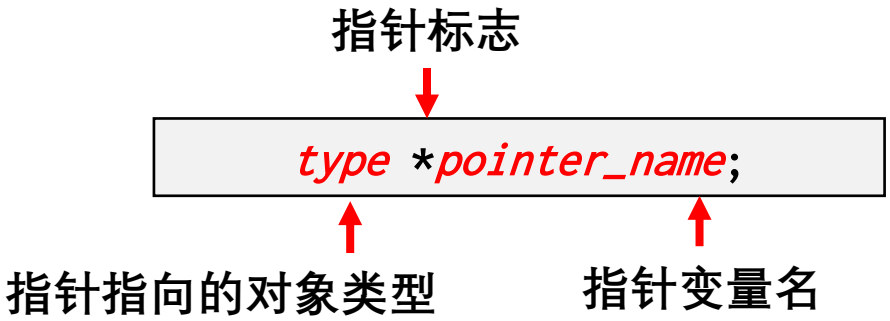
a和b是全新的变量，值从x和y拷贝而来，如果参数很大，拷贝开销不可接受

基于地址的访问方式无需拷贝

传递参数的地址和类型（宽度）
函数内基于地址访问

指针：记录地址和类型的对象

指针是复合类型：从基本/自定义类型拓展而来



一个指向整型变量的整型指针

```
int a;
int *p = &a;
```

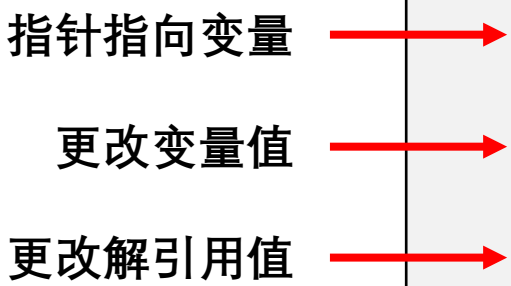
均合法

```
int* p = &a;
int * p = &a;
```

```
int* p, q;
int *p, q;
```

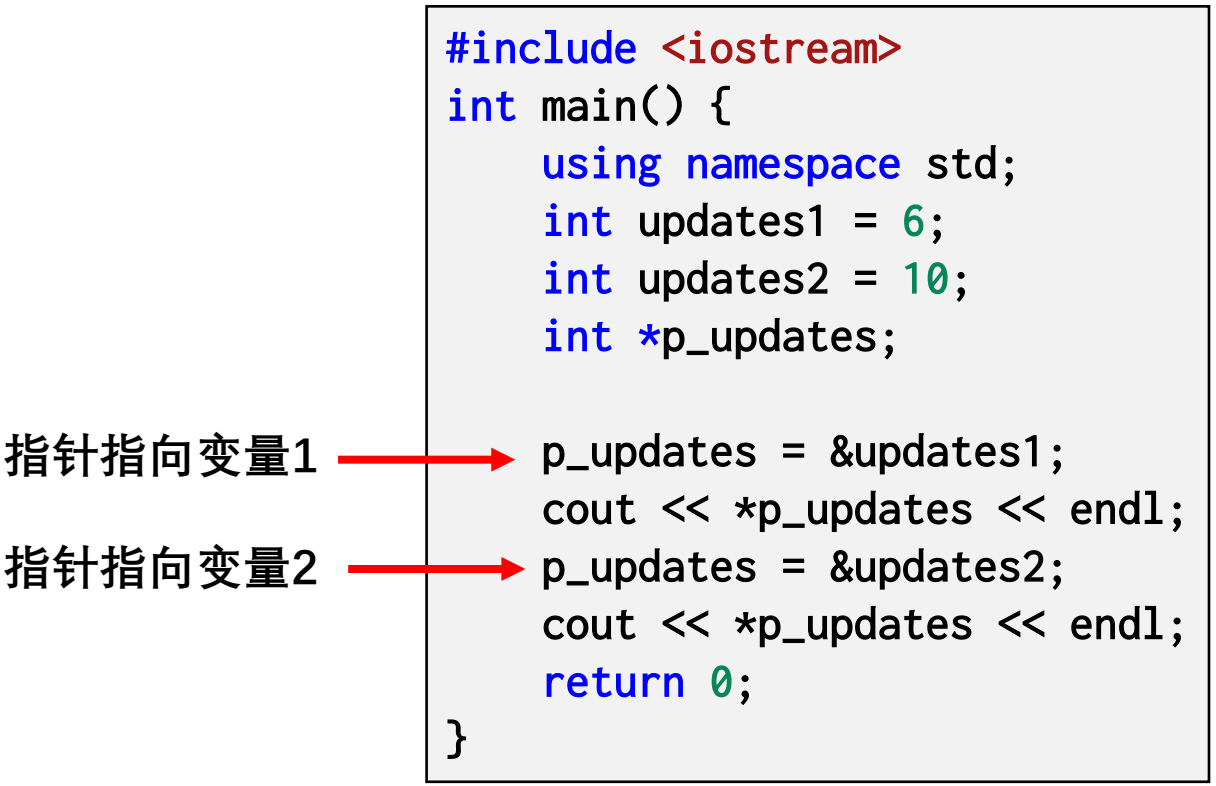
p为整型指针，q为整型

使用指针 - 1




```
#include <iostream>
int main() {
    using namespace std;
    int updates = 6;
    int *p_updates;
    p_updates = &updates;
    cout << updates << " " << *p_updates << endl;
    updates = 5;
    cout << updates << " " << *p_updates << endl;
    *p_updates = 4;
    cout << updates << " " << *p_updates << endl;

    return 0;
}
```

使用指针 - 2


指针指向变量1
指针指向变量2
#include <iostream>
int main() {
using namespace std;
int updates1 = 6;
int updates2 = 10;
int *p_updates;
p_updates = &updates1;
cout << *p_updates << endl;
p_updates = &updates2;
cout << *p_updates << endl;
return 0;
}

使用指针 - 3

错误！不允许直接修改某个变量的地址


```
#include <iostream>
int main() {
    using namespace std;
    int updates1 = 6;
    int updates2 = 10;
    &updates1 = &updates2;
    cout << updates1 << endl;
    return 0;
}
```

指针 = 地址，可以修改指针，但不能修改地址？

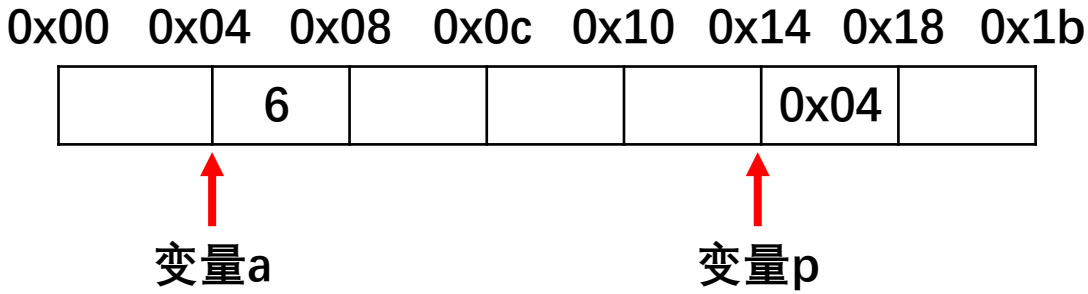
永远只能修改值，不能修改地址

深入指针的内存模型 - 1

指针是个对象，也有自己独立的存储位置

指针 = 地址 （×），指针的值是个地址（√）

```
int a = 6;
int *p;
p = &a;
```



取地址操作：获得a的地址，右值！

指针赋值：将a的地址赋给p，p的值变为a的地址

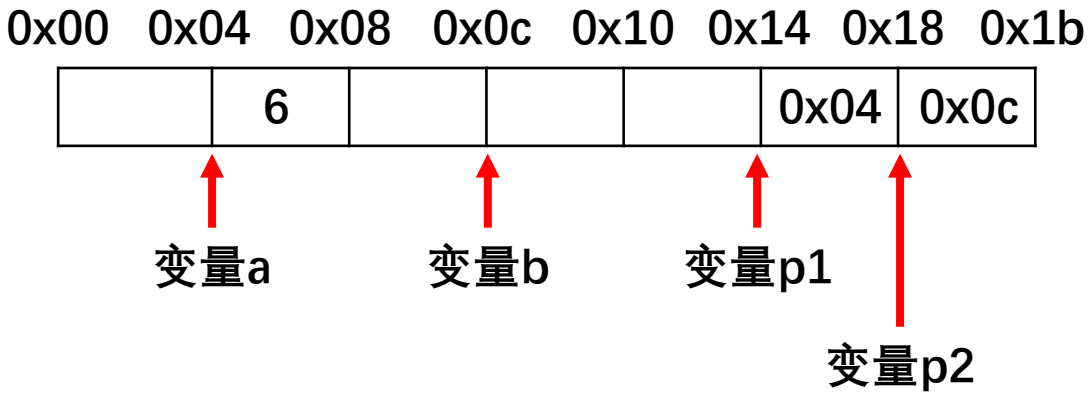
解引用操作：获取p的值所示位置的值，左值！

深入指针的内存模型 - 2

指针的值是地址 → 指针对象的长度是确定的

不同指针类型仅影响解引用过程

```
int a = 6;
int *p1 = &a;

double b = 1.0;
double *p2 = &b;

cout << *p1 << endl;
cout << *p2 << endl;
```



对整型指针解引用，按照整型理解

对浮点型指针解引用，按照浮点型理解

指针的特殊情况

仅声明指针，指针的值为不确定（野指针）

不确定的值 →

极端危险！对不确定的地址进行解引用 →

```
int *p;

*p = 10;
cout << *p << endl;
```

空指针：明确不指向任何地址的指针

明确为空 →

!p为真 →

对p进行赋值 →

输出a的值 →

```
int a = 5;
int *p = nullptr;
while (!p) {
    p = &a;
}
cout << *p << endl;
```

指针的赋值操作 - 1

与普通变量一样，相同类型的指针可以赋值

现在p和q中的值均为a的地址
p和q均指向a

效果等同于a=5和*p=5

错误，m不能存储整型变量地址

错误，n也不行

```
int a = 6;
int *p = &a;
int *q = p;

*q = 5;

double *m = &a;
double *n = p;
```

不同类型的指针可以强行转换

错误，整型不是整型指针（地址）

显式转换为地址

危险、莫名其妙但语法正确的操作

```
int *pt = 0xb8000000;
pt = (int *) 0xb8000000;
double b = 0.0;
pt = (int *)&b;
```

指针的赋值操作 - 2

void *指针，通用指针类型

void*可以指向任何类型对象

但是不能解引用，不知道对象类型

```
double obj = 3.14, *pd = &obj;
void *pv = &obj;
pv = pd;

cout << *pv << endl;
```

一种古老的多态实现方式

返回一片未定义类型的内存空间

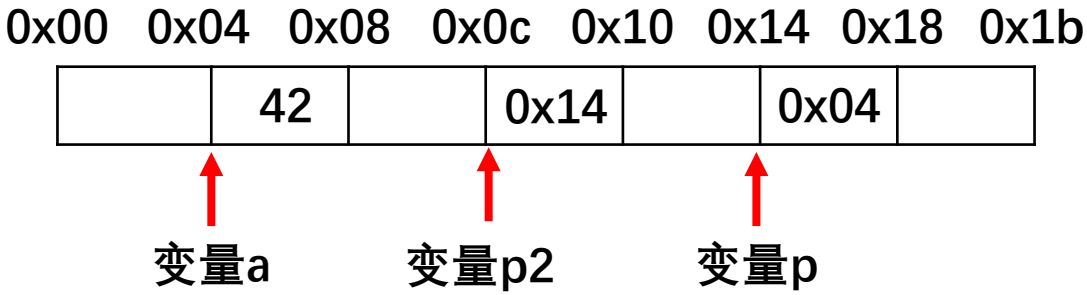
接受多种类型的参数

指针的指针

指针存储地址，指针的指针也存储地址

指针的指针 →

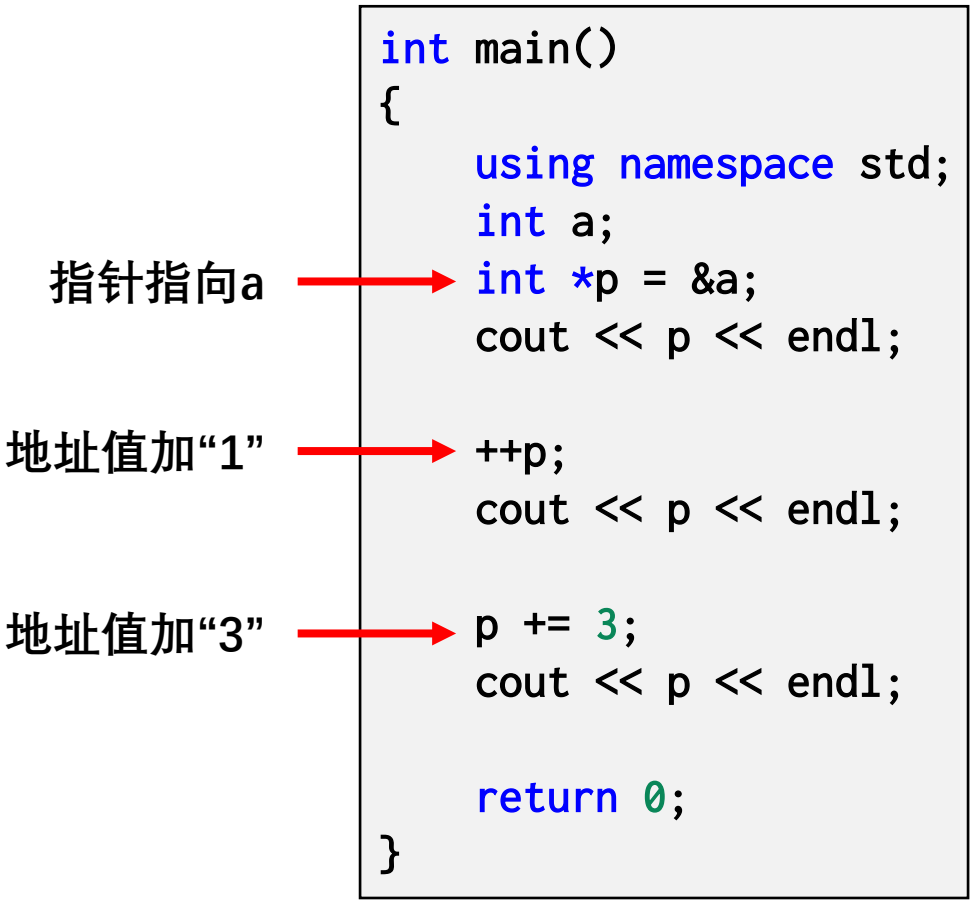
```
int a = 42;
int *p = &a;
int **p2 = &p;
```



重访数组和字符串

指针算术 - 1

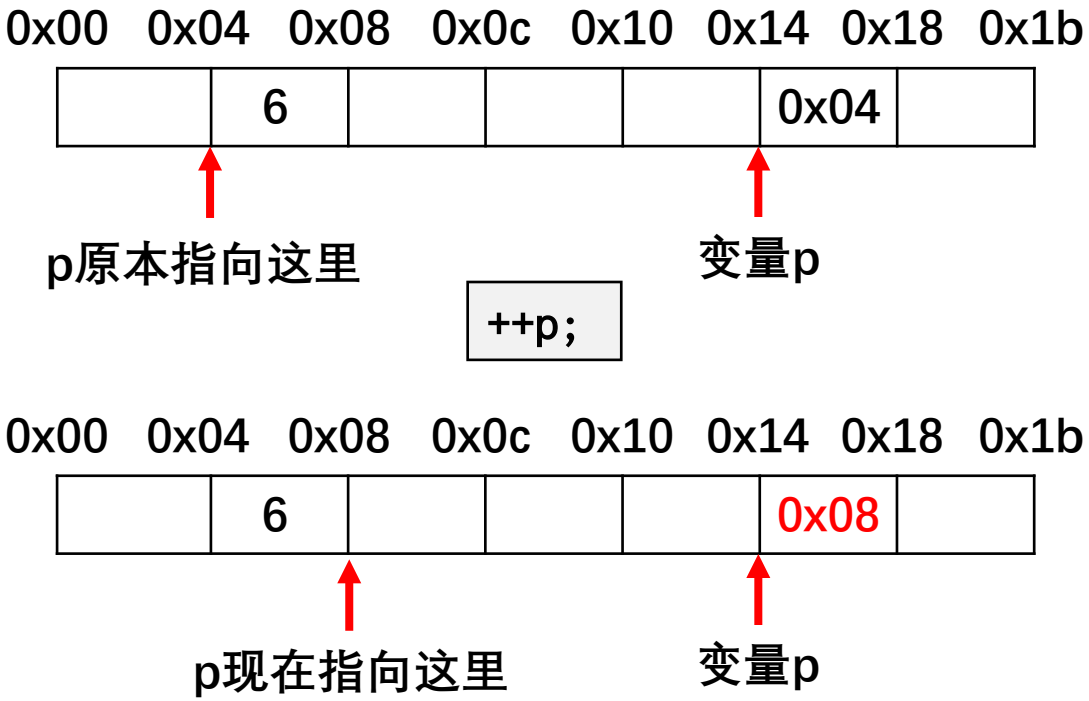
允许对指针进行整数加减运算




```
int main()
{
    using namespace std;
    int a;
    int *p = &a;
    cout << p << endl;

    ++p;
    cout << p << endl;

    p += 3;
    cout << p << endl;

    return 0;
}
```

指针算术 - 2

指针的整数运算，实际上是地址按照类型宽度进行变化



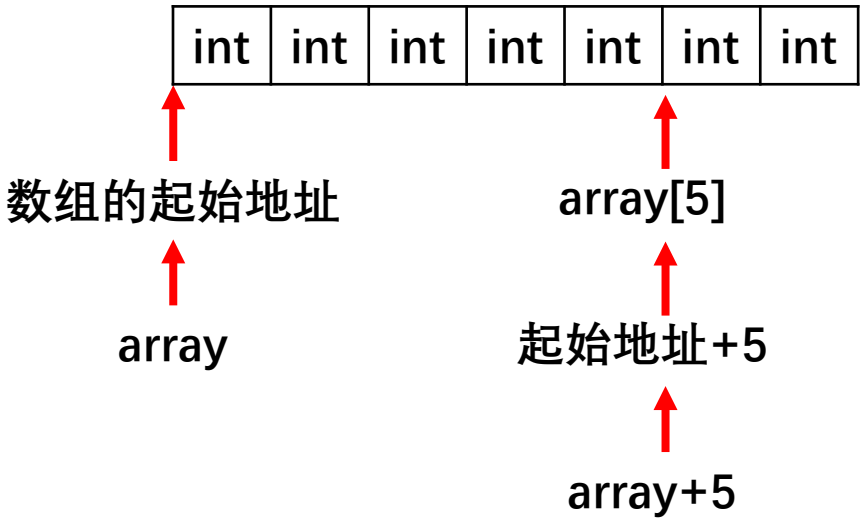
为什么需要这么奇怪的指针算术?

指针与数组的关系 - 1

指针的整数运算是地址按照类型宽度进行变化

数组是一片地址连续的空间，以类型宽度分割

```
int array[7];
```



指针与数组的关系 - 2

数组名绑定在数组的首地址 – 右值!

array首地址

第一个元素的地址，与首地址相同

第三个元素的地址

```
#include <iostream>
int main()
{
    using namespace std;
    int array[3] = {1, 2, 3};

    cout << array << endl;

    cout << &array[0] << endl;
    cout << &array[2] << endl;
    cout << array+2 << endl;

    return 0;
}
```

指针与数组的关系 - 3

对数组的访问，实际上是指针访问的语法糖

array[i]与*(array+i)完全等价

甚至与i[array]完全等价

但千万别这么用

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

    int array[3] = {1, 2, 3};

    for (int i = 0; i < 3; ++i) {
        cout << *(array + i) << endl;
        cout << array[i] << endl;
    }

    return 0;
}
```

指针与数组的关系 - 4

数组名是右值，但可以用指针变量来灵活访问

使用p作为游标在数组中随意移动

按照地址访问array的最后一个元素

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

    int array[3] = {1, 2, 3};

    int *p = array;
    cout << *(++p) << endl;
    cout << p[1] << endl;

    return 0;
}
```

指针和字符串的关系 - 1

字符串在底层以字符数组的方式存储

字符串常量是右值，需要使用const指针存地址

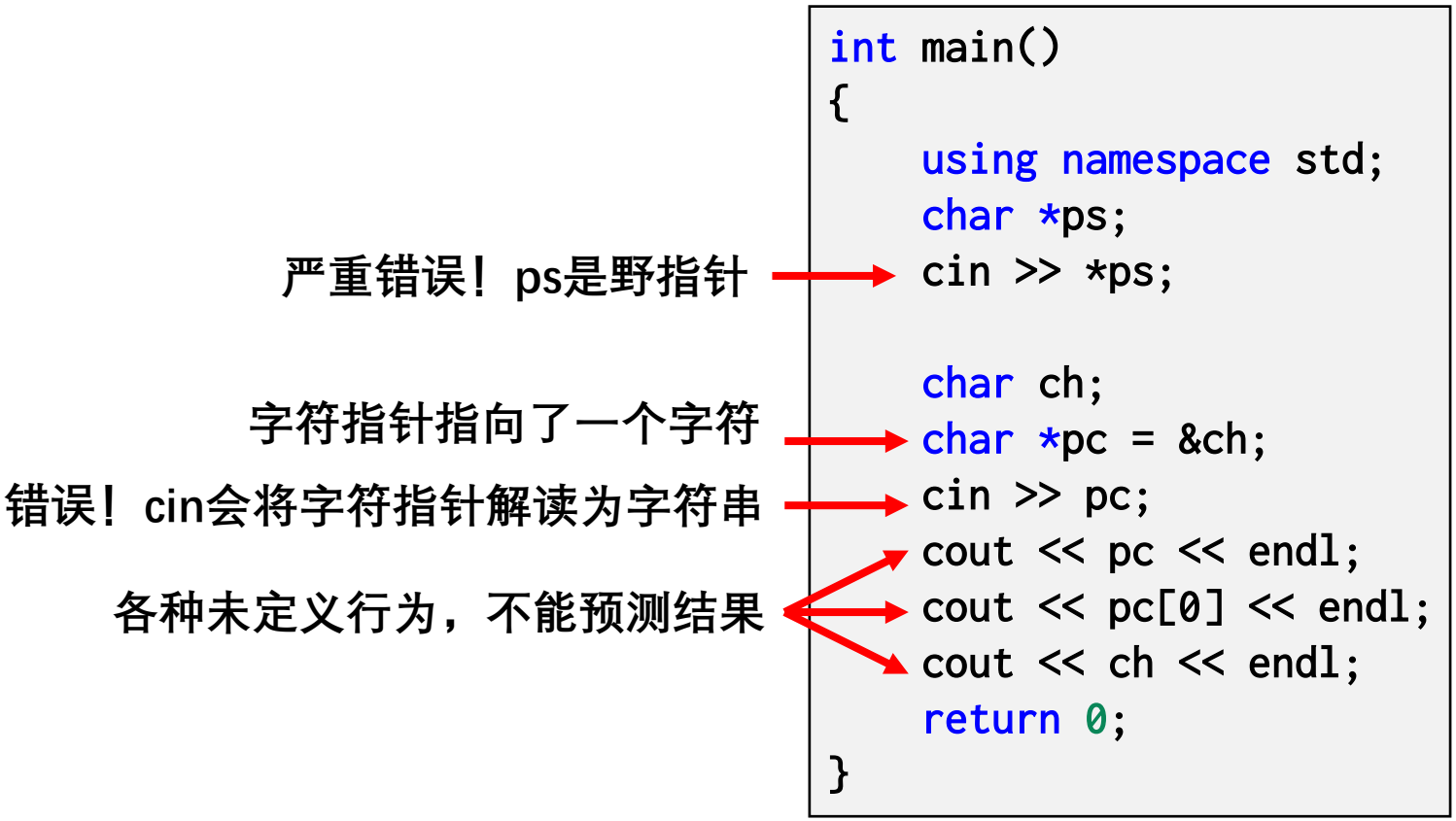
animal和bird都是字符对象地址，cout均看作字符串

ps指针指向animal

```
#include <iostream>
int main()
{
    using namespace std;
    char animal[20] = "bear";
    const char *bird = "wren";
    char *ps;
    cout << animal << " and " << bird << endl;
    cout << "Enter a kind of animal: ";
    cin >> animal;
    ps = animal;
    cout << ps << endl;
    return 0;
}
```

指针和字符串的关系 - 2

以字符指针作为字符串时的陷阱




```
int main()
{
    using namespace std;
    char *ps;
    cin >> *ps;

    char ch;
    char *pc = &ch;
    cin >> pc;
    cout << pc << endl;
    cout << pc[0] << endl;
    cout << ch << endl;
    return 0;
}
```

内容总结

指针：存储地址的类型，固定宽度

- 指向某个对象：值为某个对象的地址
- 野指针、空指针、指针赋值
- 变量与地址的绑定关系不变，变量的值可以变

指针视角下的数组：

- 数组名 = 数组首地址
- 访问数组元素 = 地址偏移