高级语言C++程序设计
指针复习

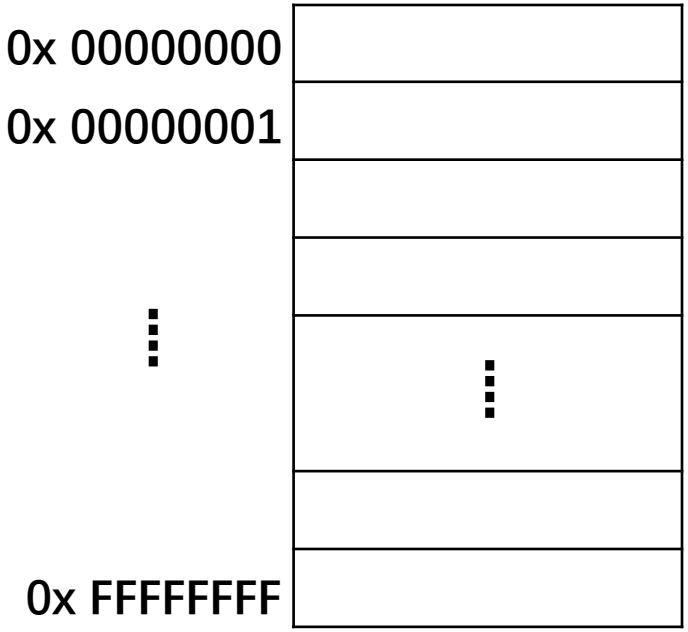
南开大学 计算机学院

2022

计算机内存

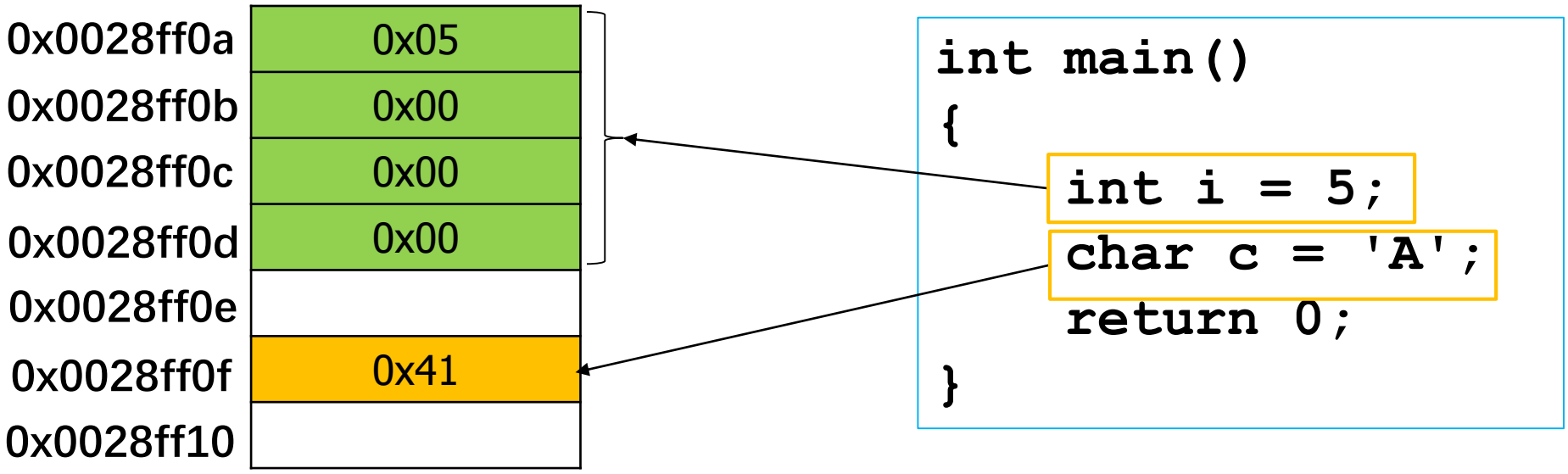
内存：计算机的存储空间，用于程序运行时保存数据 (程序指令、变量等)

- ✓ 以字节(Byte)为计量单位
- ✓ 每个字节有一个地址
- ✓ 地址用32 bits表示：例如，0x 0025f758
- ✓ 内存地址的范围是 $0 \sim 2^{32}-1$



内存地址

变量在内存中的地址

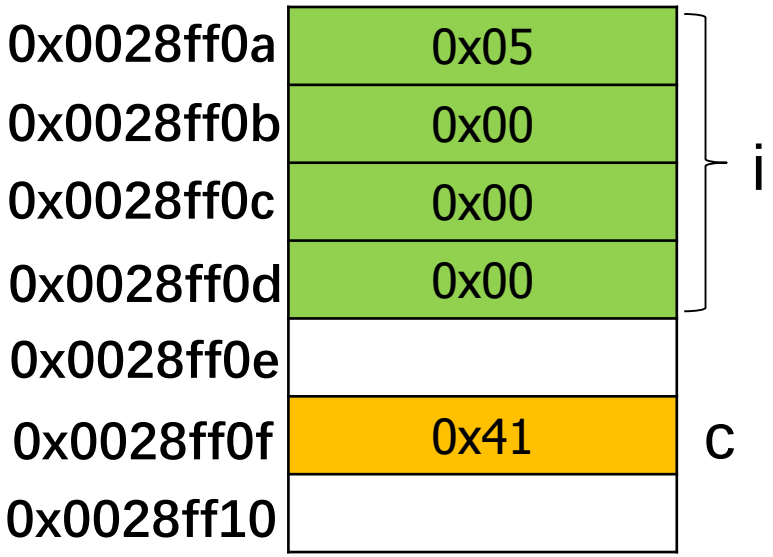


变量 i 在内存中的（起始）地址为 0x0028ff0a

变量 c 在内存中的地址为 0x0028ff0f

内存地址

如何得到变量的地址？ & <变量>



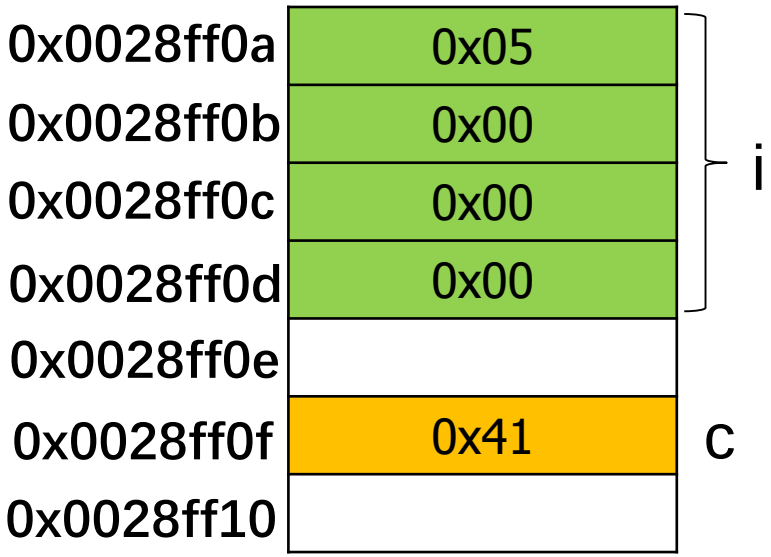

```
int main()
{
      int i = 5;
      char c = 'A';
      cout<<&i<<endl;
      cout<<(void*) &c<<endl;
      return 0;
}
```

取变量 i 的地址，输出 0x0028ff0a

取变量 c 的地址，输出 0x0028ff0f

内存地址

如何从地址得到变量？ *<地址>




```
int main()
{
    int i = 5;
    char c = 'A';
    cout<<*(&i)<<endl;
    cout<<*(&c)<<endl;
    return 0;
}
```

输出 5

输出 A

内存地址

用什么**数据类型**表示内存地址？

指针

最终方案:
<变量类型> + *

```
#include <iostream>
using namespace std;
int main() {
    int i = 5;
    char c = 'A';
    int * p = &i; //p表示变量i的地址
    char * q = &c; //q表示变量c的地址
    cout<<p<<" "<<(void *)q<<endl; //输出地址
    cout<<*p<<" "<<*q<<endl; //输入对应变量的值
    return 0;
}
```

指针

指针是一种新的数据类型，用来表示内存地址

指针定义：

<数据类型> * 指针变量名;

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

p为int*型变量，p存储着int型变量i的地址，由p可以找到变量i，因此，也称p为指向i的指针

指针占4个字节，32 bits

指针的基本操作

指针初始化

未进行初始化，也称悬挂指针

```
int *a;//虽然未初始化，也占用4个字节空间
```

初始化为 0 或者 NULL，指向地址0，不可访问空间

```
int *a = 0;
int *b = NULL; //等同于0
cout<<*a<<*b<<endl;//错误！0地址不可访问
```

初始化为已定义变量的地址

```
int a = 0;
int *b = &a;
```

指针初始化

指针变量的数据类型与其指向的数据类型必须一致

```
int a;
int *p1 = &a; //ok
char *p2 = &a; //error, 类型不一致
```

数据类型不一致时，可通过强制类型转换

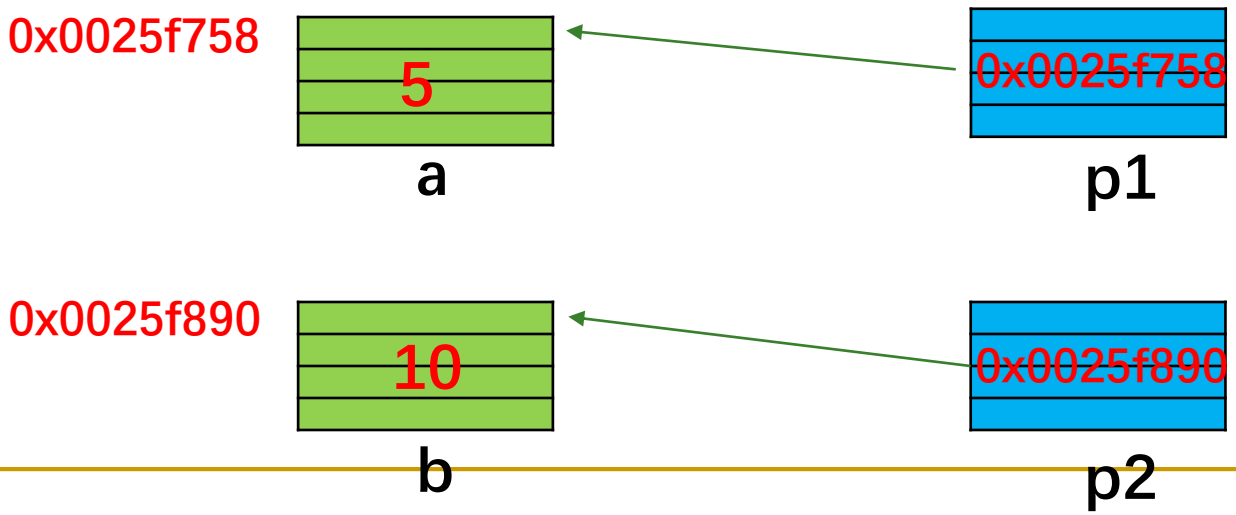
```
int a = 0x61626364;
int *p1 = &a; //ok
char *p2 = (char*)&a; //强制类型转换
```

指针基本操作

通过间接访问运算符访问指针所指向的变量

* <指针表达式>

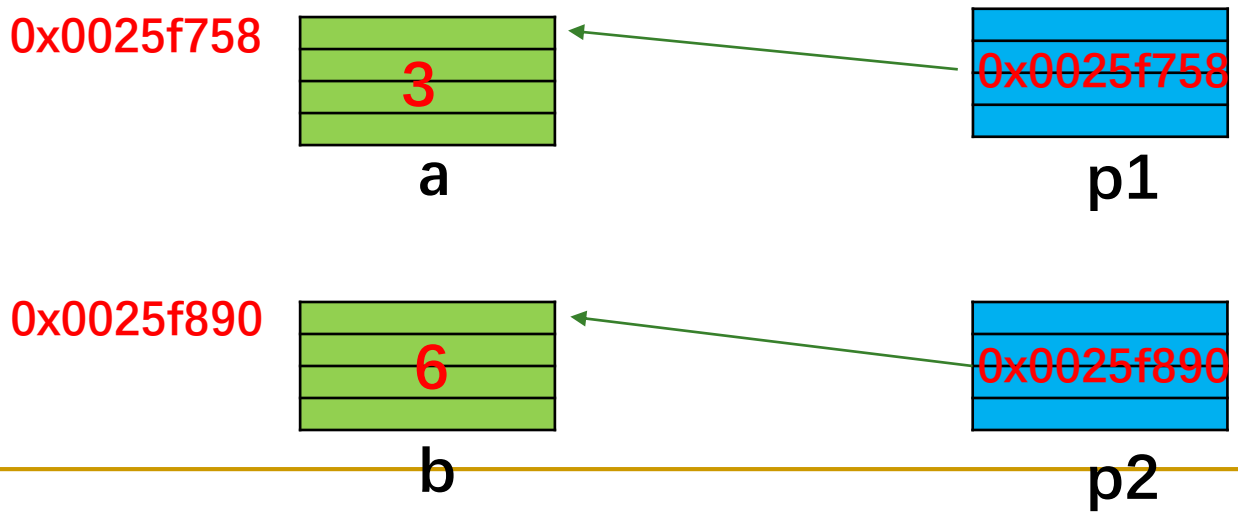
```
int a = 5, b = 10;
int  *p1, *p2;
p1 = &a; p2 = &b;
cout<<a<<b<<*p1<<*p2;
```



指针基本操作

```
int a = 5, b = 10;
int  *p1, *p2;
p1 = &a; p2 = &b;
*p1 = 3; *p2 = 6;
cout<<a<<b<<*p1<<*p2;
```

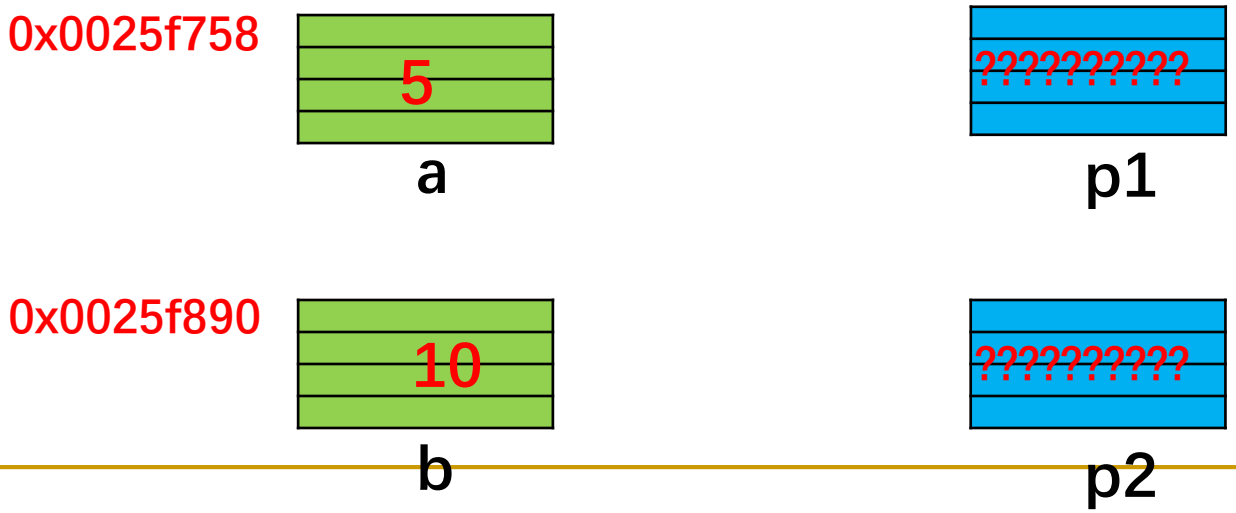
*p1对应的是变量a，因此可以作为左值



指针基本操作

```
int a = 5, b = 10;
int  *p1, *p2;
*p1 = 3; *p2 = 6;
```

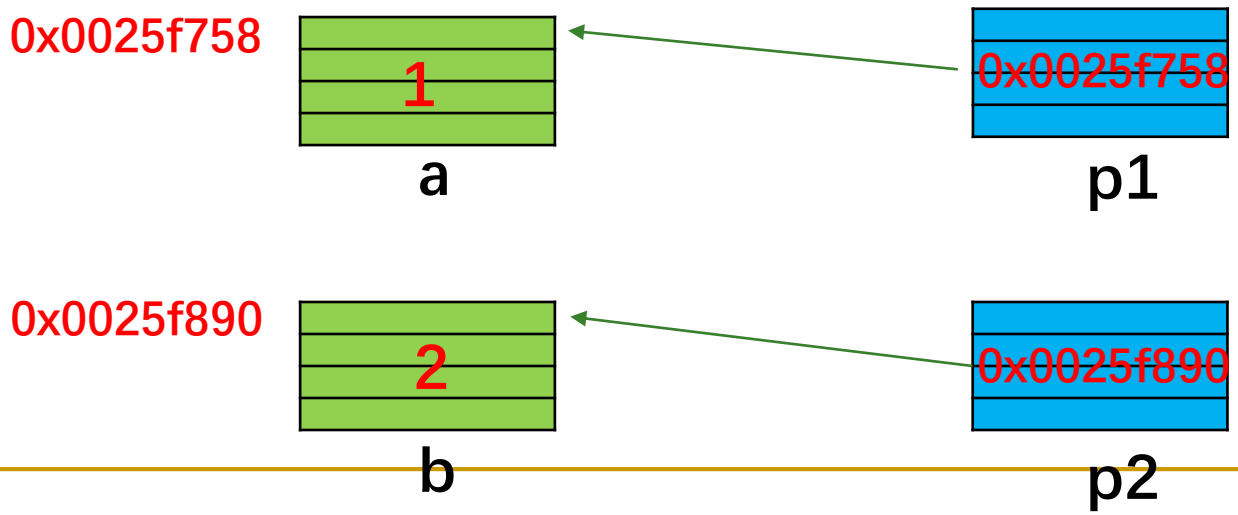
Fatal Error! 作为左值之前必须初始化！否则指向未知区域



指针基本操作

```
int a = 5, b = 10;
int  *p1, *p2;
p1 = &a; p2 = &b;
a = 1; b = 2;
cout<<a<<b<<*p1<<*p2;
```

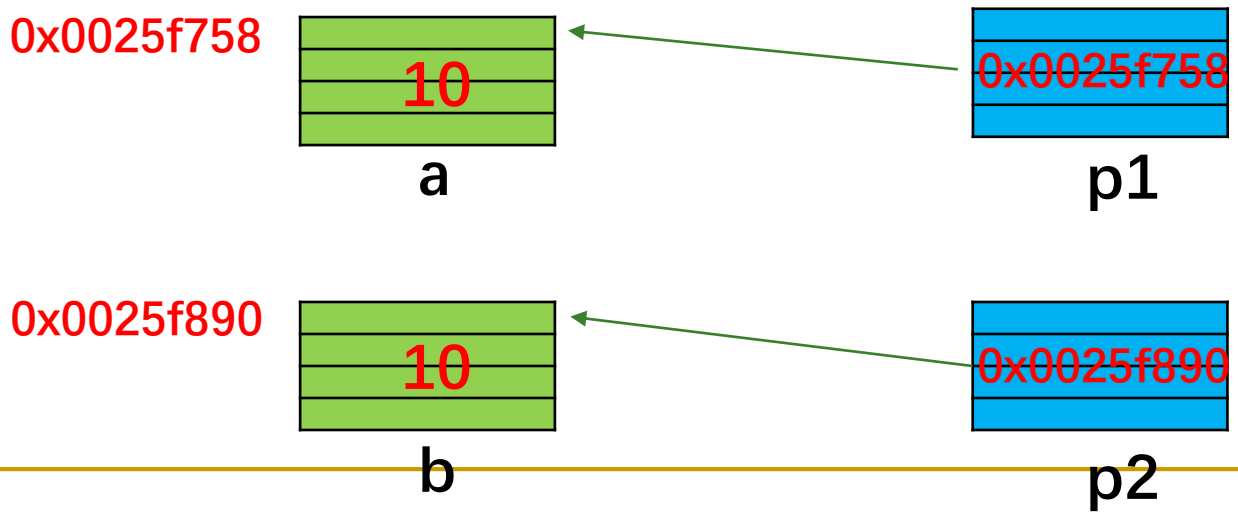
变量改动与指针无关



指针基本操作

```
int a = 5, b = 10;
int  *p1, *p2;
p1 = &a; p2 = &b;
*p1 = *p2;
cout<<a<<b<<*p1<<*p2;
```

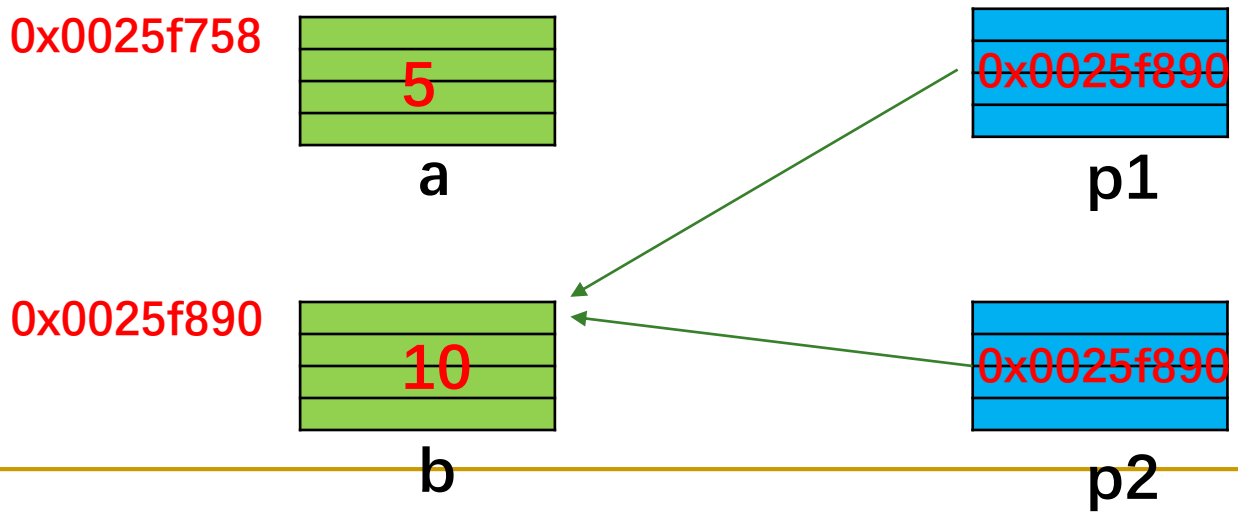
*p1作为左值，*p2作为右值



指针基本操作

```
int a = 5, b = 10;
int  *p1, *p2;
p1 = &a; p2 = &b;
p1 = p2;
cout<<a<<b<<*p1<<*p2;
```

p1、p2本身是变量，可以作为左值或右值



指针运算

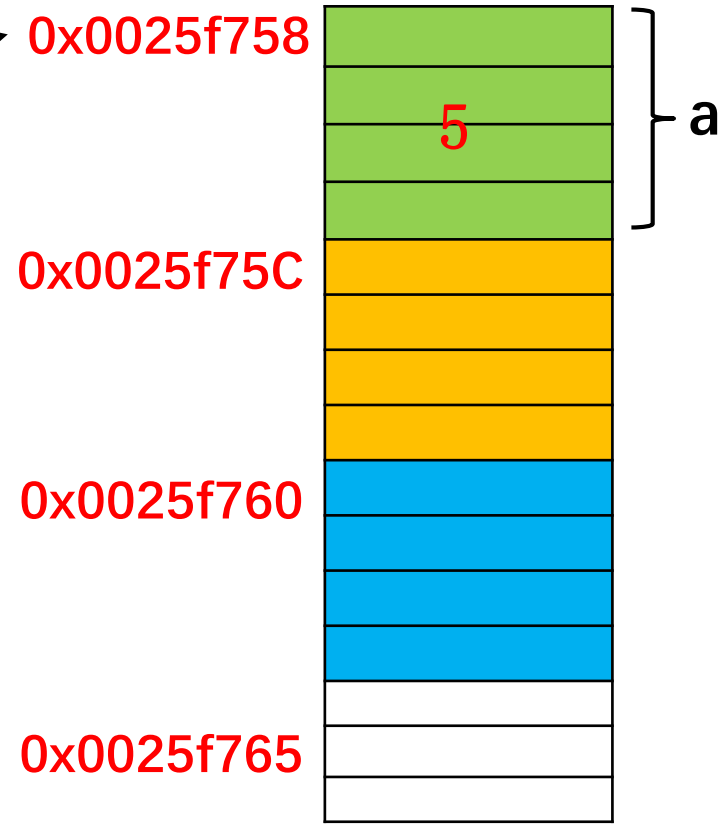
指针的算术运算

指针变量可进行算术运算，包括+、-、++、--等

```
int a = 5;
int  *p = &a;
p = p + 1; //p+1*4
```

指针+整数 =
指针+(整数x所指类型的字节数)

```
p ++; //p+1*4
cout<<*(p-3);
```

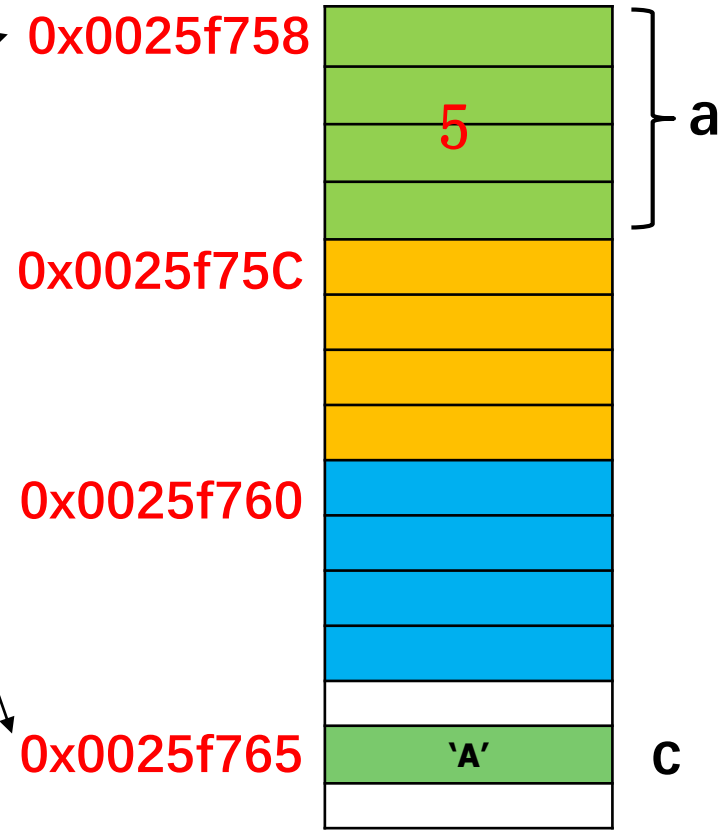


通过指针的算术运算，可以访问未知区域！非常危险！

指针的算术运算

指针变量可进行算术运算，包括+、-、++、--等

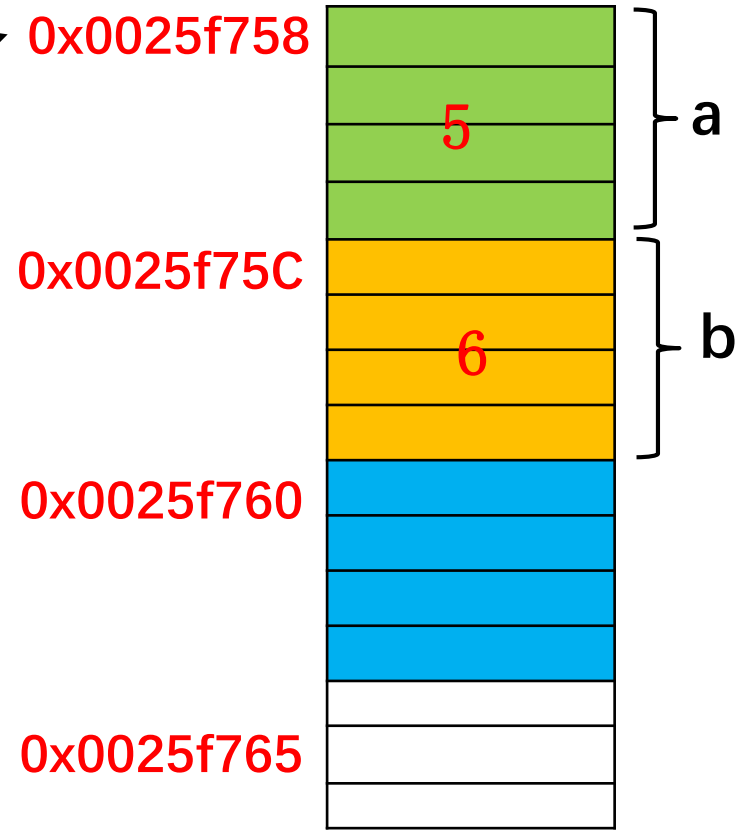
```
int a = 5;
int  *p = &a;
p = p + 2; //p+2*4
char c = 'A';
char *p1 = &c;
p1 = p1 - 5; //p1-5*1
```



指针变量之间的运算

指针变量之间也可以运算，包括相减、比较等

```
int a = 5, b = 6;
int  *p1 = &a;
int *p2 = &b;
cout<<p2-p1;
```



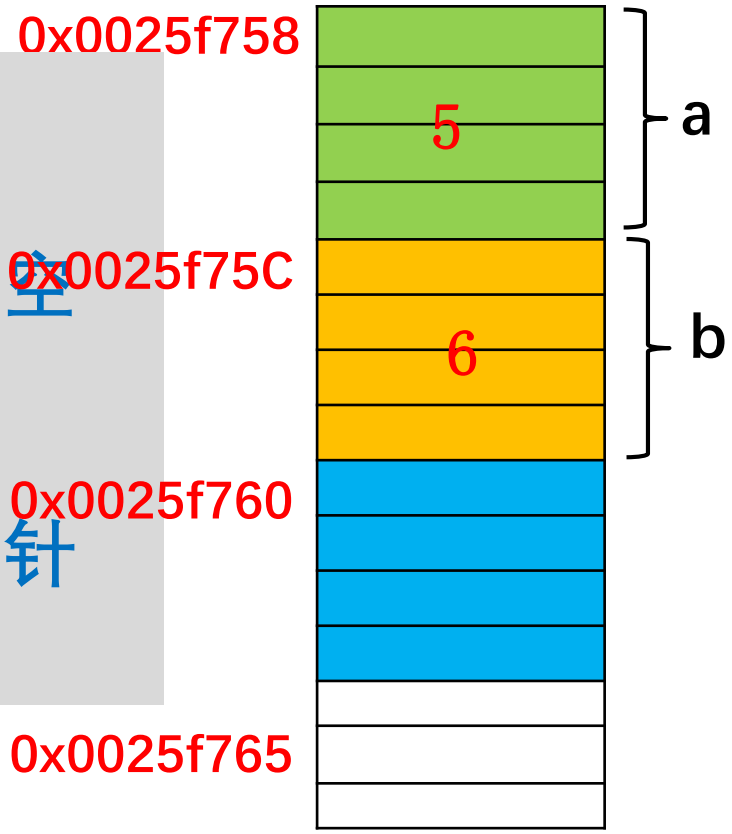
同类型指针相减等于两个指针之间所指类型数据的个数

p2减p1等于p2和p1之间int型数据的个数，即1

指针变量之间的运算

指针变量之间也可以运算，包括相减、比较等

```
int a = 5, b = 6;
int  *p1 = &a;
if(p1 != NULL)//判断是否为空
    *p1 = 10;
int *p2 = &b;
while(p1 < p2)//比较两个指针
    p1++;
```



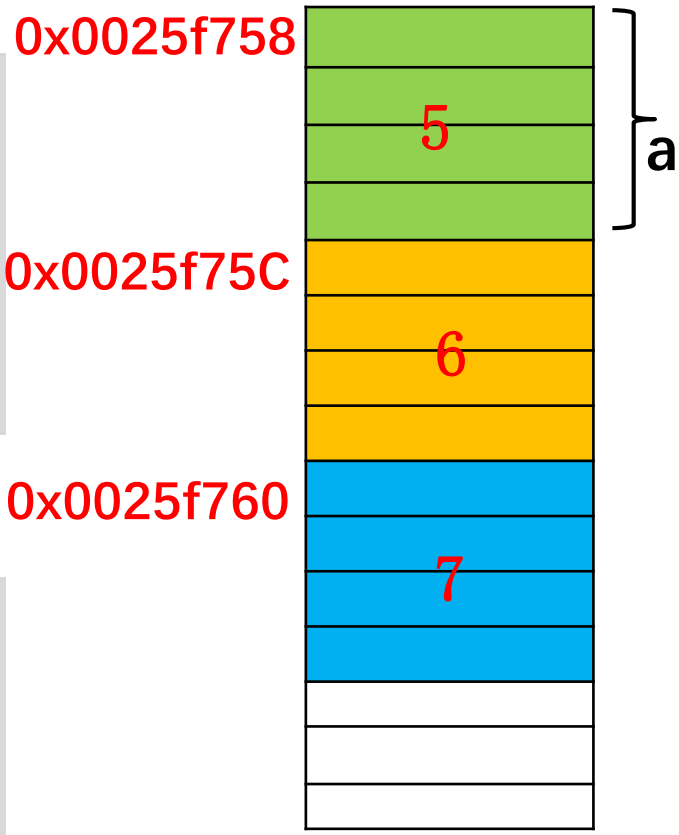
只有相同类型指针才能够进行比较！

指针变量的下标访问

*(<指针>+i) 等价于<指针>[i]

```
int a = 5;
int  *p = &a;
cout<<*p<<*(p+1)<<*(p+2);
cout<<p[0]<<p[1]<<p[2];
```

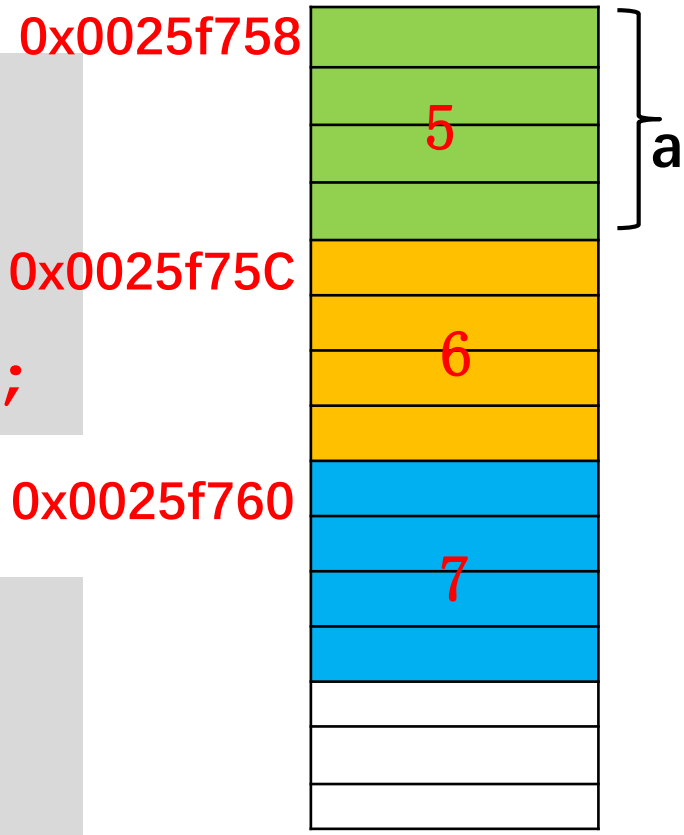
```
p[0] == *p
p[1] == *(p+1)
p[2] == *(p+2)
```



指针变量的下标访问

(<指针>+i) 等价于 &<指针>[i]

```
int a = 5;
int  *p = &a;
cout<<p<<(p+1)<<(p+2);
cout<<&p[0]<<&p[1]<<&p[2];
```

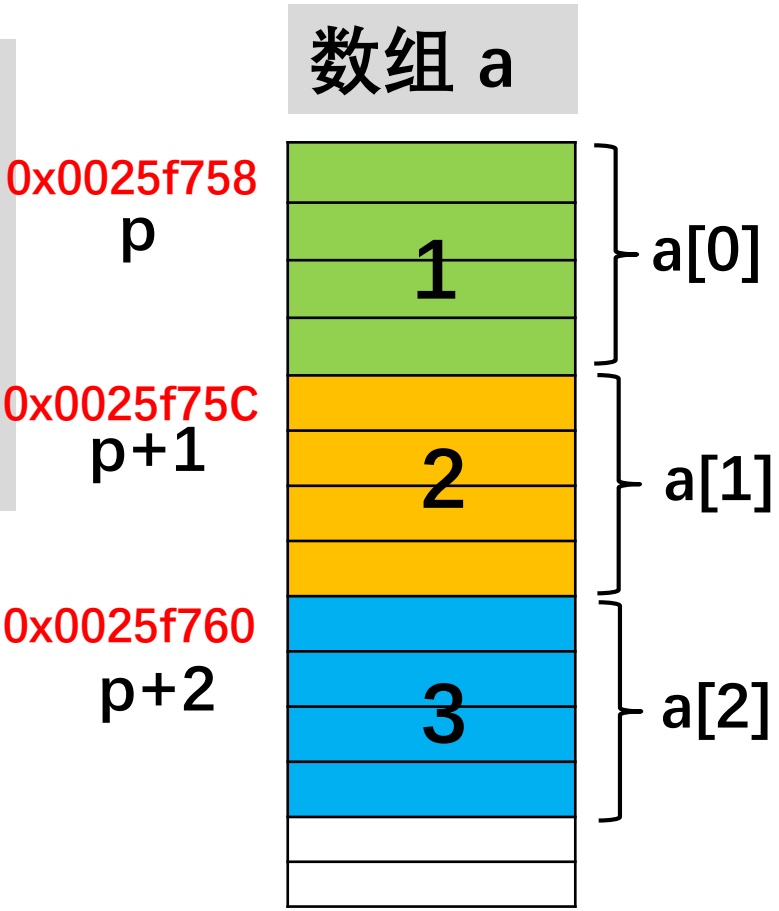



```
&p[0] == p
&p[1] == (p+1)
&p[2] == (p+2)
```

指针与数组

指针与数组

```
int a[3] = {1, 2, 3};
int *p = &a[0];
cout<<*p<<*(p+1)<<*(p+2);
cout<<p[0]<<p[1]<<p[2];
cout<<a[0]<<a[1]<<a[2];
```

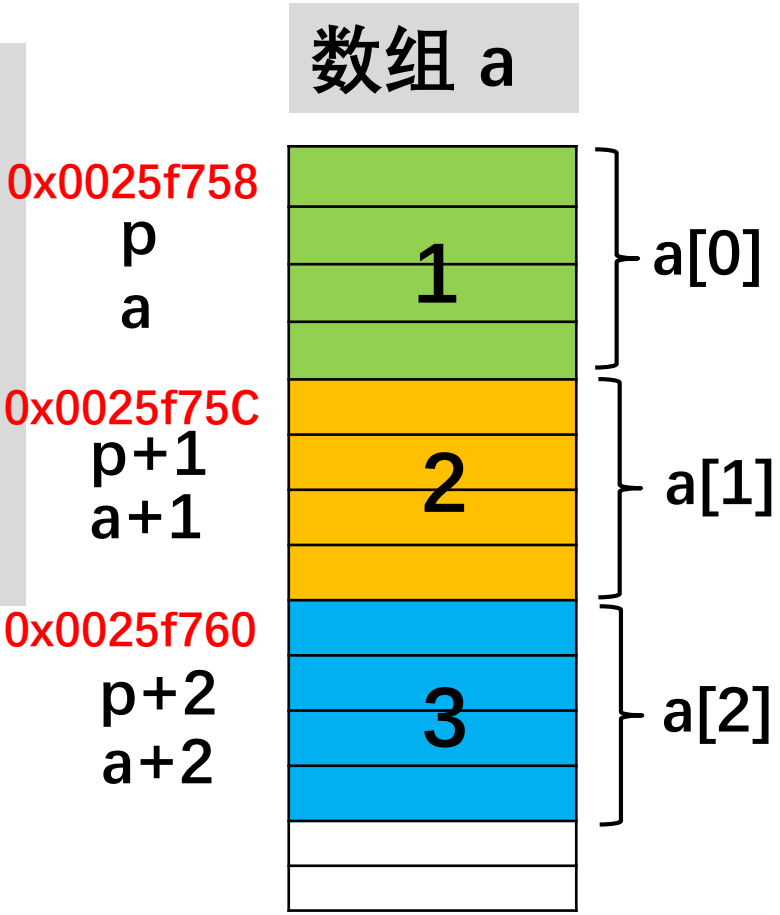


p是指向数组首元素的指针，
p实现与数组名a相同的功能

数组名其实是指向数组首元素的指针，即 a == &a[0]

指针与数组

```
int a[3] = {1, 2, 3};
int *p = &a[0];
cout<<*p<<*(p+1)<<*(p+2);
cout<<*a<<*(a+1)<<*(a+2);
cout<<p[0]<<p[1]<<p[2];
cout<<a[0]<<a[1]<<a[2];
```



数组名其实是指向数组首元素的指针，即 a == &a[0]

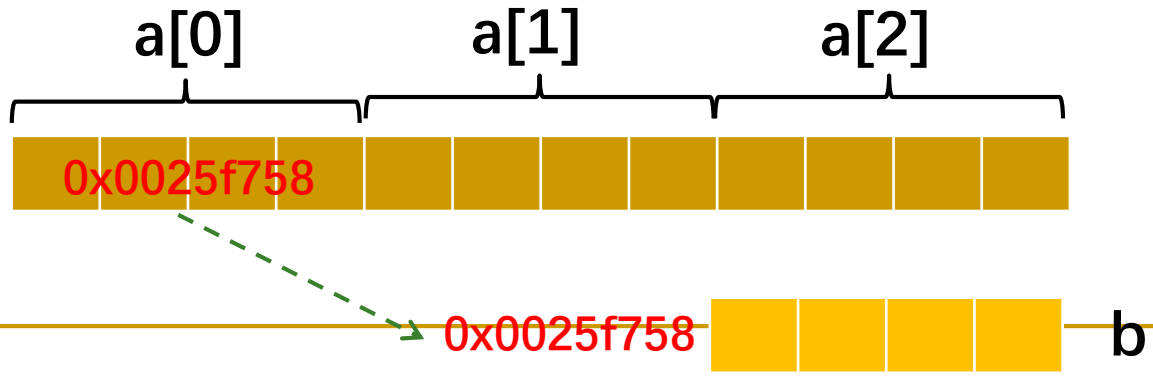
指针数组

指针数组：数组元素为指针类型

```
int * a[3] ; //a为数组，元素类型为 int *
```

分析：[]优先级高于*, a和[3]先结合，表明a是数组，剩余部分（即int *）表示元素类型

```
int *a[3]; //定义数组，未初始化
int b = 2;
a[0] = &b ; //数组第一个元素赋值
```



数组指针

数组指针：指向数组的指针

```
int(*a)[2];//a为指针，指向数组类型int[2]
```

分析：(*a)表明a是指针，剩余部分（即int [2]）
表明a指向的类型（大小为2的整型数组）

```
int b[2];
a = &b; //此时b代表整个数组
```

a+1 等价于 a + 1*数组b的大小

数组指针

数组指针：指向数组的指针

```
int(*a)[2];//a为指针，指向数组类型int[2]
int b[2];
a = &b;
```


```
cout<<&b[0];
cout<<b;
cout<<&b;
cout<<a;
```

&b[0]、b、&b和a的值都是0x0025f758

数组指针

数组指针：指向数组的指针

```
int(*a)[2];//a为指针，指向数组类型int[2]
int b[2];
a = &b;
```


```
a = b; //ok?
```

错误，a是指向int [2]型的指针，b是指向int型的指针

```
a = &b[0];//ok?
```

错误，a是指向int [2]型的指针，&b[0]是int型变量的地址

总结

- ✓ 数组名是指向数组“首”元素的常量指针

  a == &a[0]

- ✓ 分析变量是数组还是指针注意优先级

  int *b[3];　　int (*b)[3];

- ✓ 两个万能变换公式

  p+i 等价于 &p[i]　　*(p+i) 等价于 p[i]

指针与字符串

用字符数组存储字符串

```
char s[] = "Hello World";
char *p = s;
```


```
cout<<p; //输出的是整个字符串,不是地址！
```

```
cout<<p+1; //ok?
```

```
cin>>p; //键盘输入新的字符串，不带空格
```

```
p[0] = 'a'; //更改数组元素
```

指针与字符串

用指针声明的字符串(字符串常量)

```
char *s = "Hello World";
```


```
cout<<s;
```

```
cin>>s; //error,常量不可更改
```

```
s[0] = 'a'; //error,常量不可更改
```

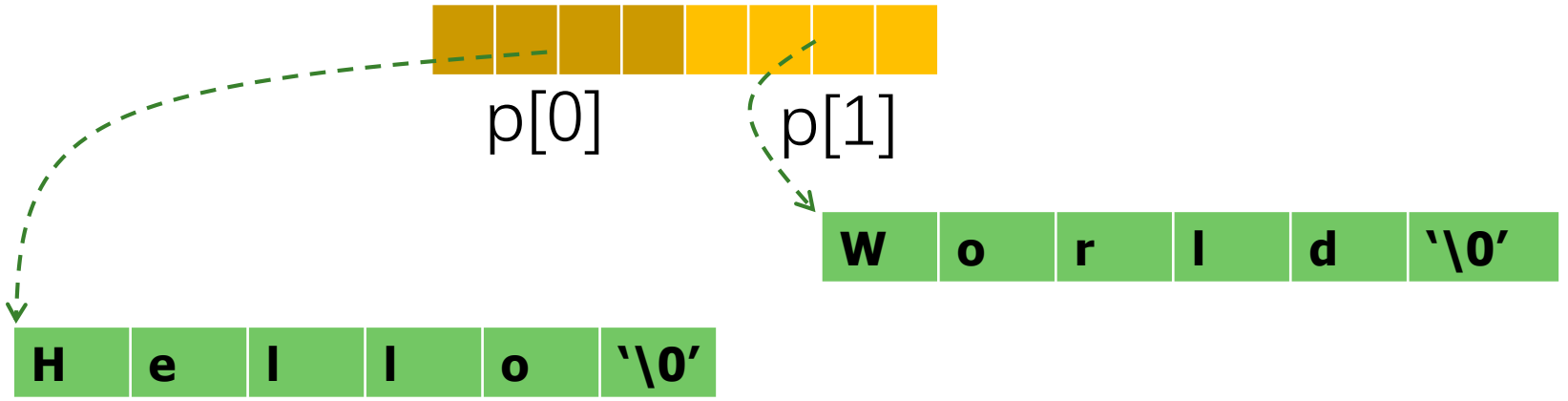
```
char c[] = "abc"
s = c// ok, s是指针变量，其值可以改
```

```
s[0] = 'a'; ok
```

指针与字符串

指针数组和字符串

```
char *p[2] = {"Hello", "World"};
```




```
cin>>p[0]; //error
p[0] = "abc";//ok, 指向另外一个字符串常量
```

p[0]、p[1]指向的字符串常量不可修改，但p[0]、p[1]本身的值可以修改！

动态分配内存

传统变量定义通过静态方式分配内存

```
int a;
int b[1000];
```

- 程序编译阶段已经确定内存大小
- 静态分配的内存在栈区，每个应用程序有一个栈，栈有大小限制（Linux系统默认为8MB）

动态分配内存

指针的主要作用是动态内存分配

动态分配一个数组：

new <数据类型> [数组大小]

```
int n, *p;
cin>>n;
p = new int[n];
p[2] = 3;
```

- new 返回数组首元素的地址，p是指向首元素的指针
- 动态分配的数组没有名字，需要通过指针p来访问
- 数组大小可以是变量！

动态分配的空间在堆区，内存多大堆就有多大！

动态分配内存

动态内存访问

用指针操作，和访问数组一样，但首先要判断是否分配成功

```
int n, *p;
cin>>n;
if(n > 1) {
     p = new int [n];//分配内存
}
if (p == NULL) return 0; //判断是否成功
```

```
for (int i = 0; i < n; i ++) {
     cin>>p[i]; //访问数组元素
}
```

动态分配内存

内存回收

动态分配的内存必须进行回收，否则内存泄漏

delete 和 delete []

```
int n, *p;
p = new int; //单个变量
delete p; //内存回收,单个变量时用delete
cin>>n;
p = new int [n]; //数组
delete []p; //内存回收,数组时用delete[]
```

new 和 delete配对
new [] 和delete []配对

动态分配内存

```
int *p;
p = new int; //动态生成一个int变量
delete p; //回收动态分配的空间

p = new int[10]; //动态生成int数组
delete p; //回收动态分配的空间
delete []p; //回收动态分配的空间
```

int等基本数据类型没有析构函数，所以delete p和delete p[]在回收动态数组的时候等价

动态分配内存

为结构体分配动态内存

```
struct student {
        int id;
        char *name;
};
```

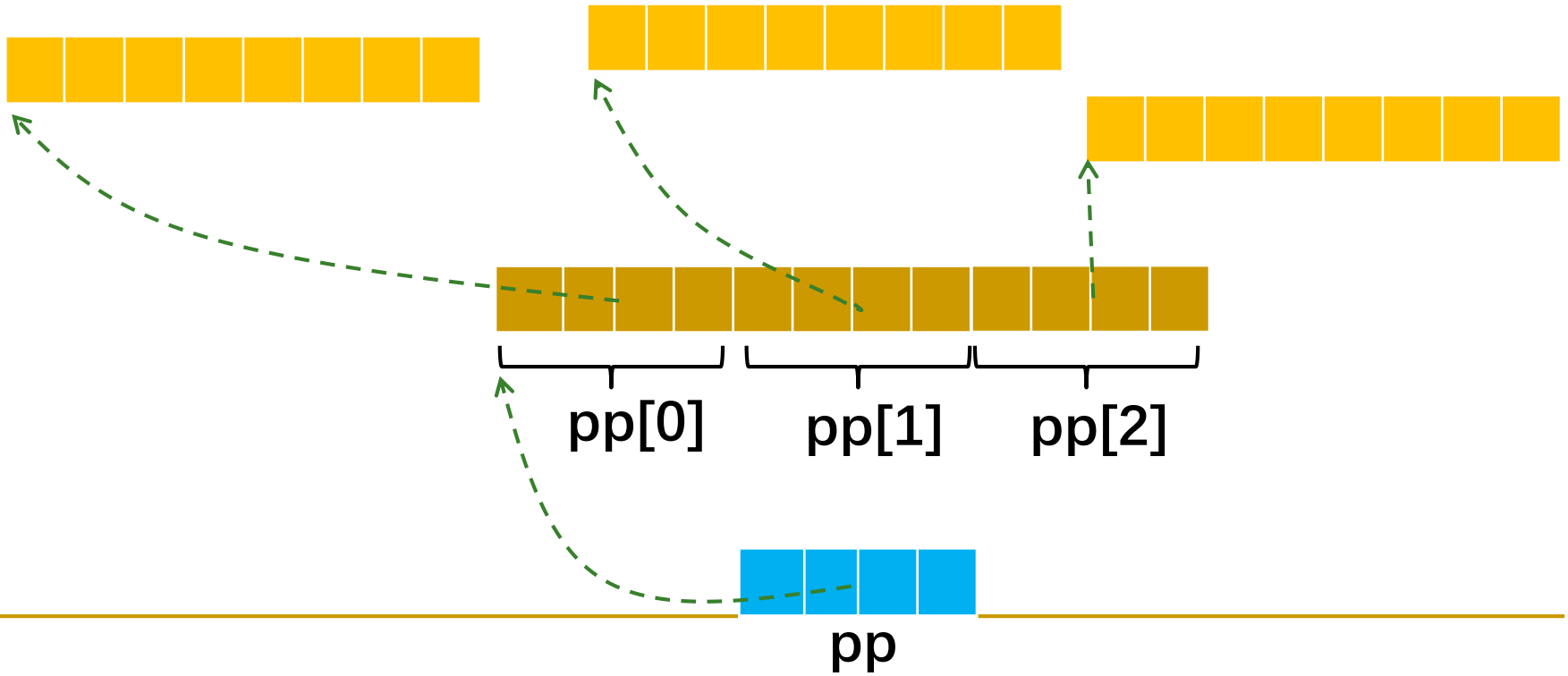
```
student *p = new student[2];
cout<<p[0].id<<p[0].name;
cout<<p->id<<p->name;
```

构体指针可以通过 -> 运算符访问结构体成员

动态分配的二维数组

动态生成m行n列的二维数组， m、n均为变量

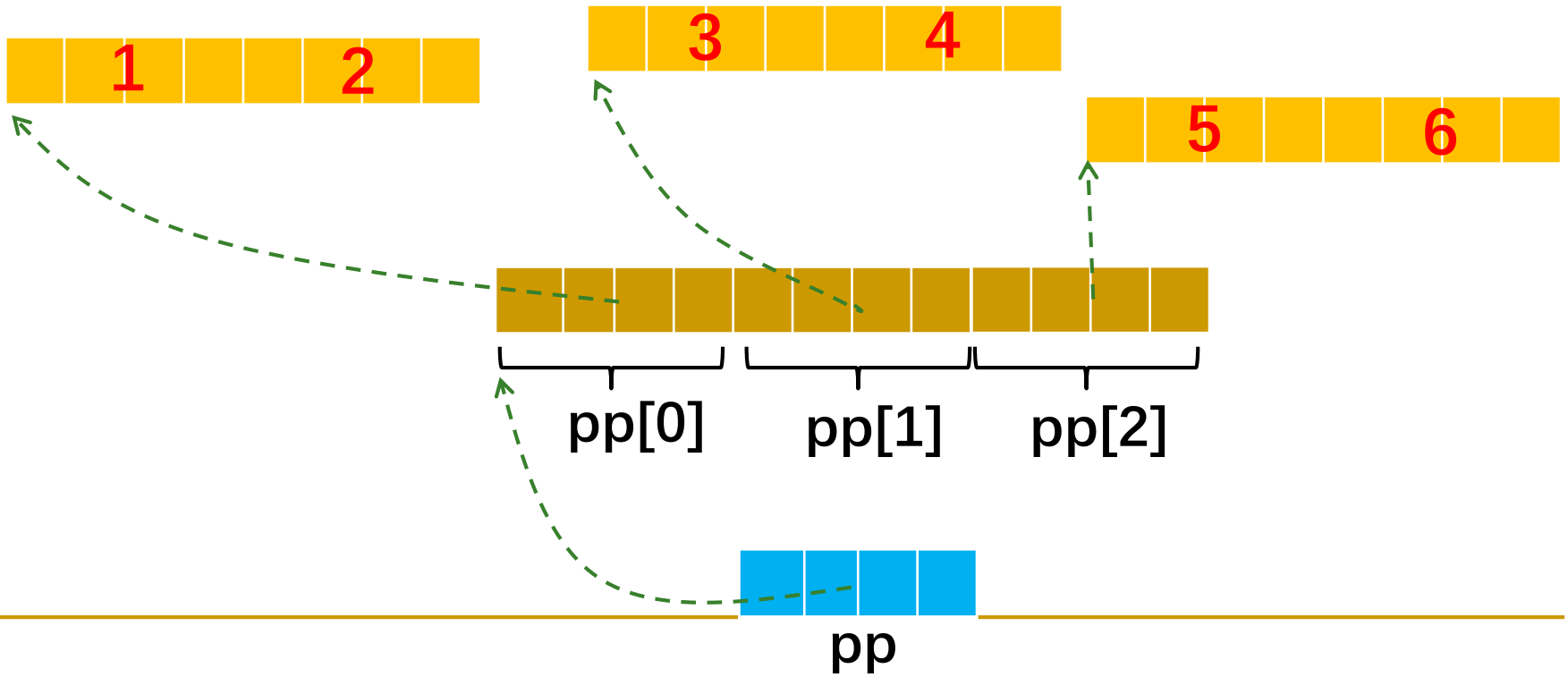
```
int **pp = new int* [m]; //m为变量
for(int i = 0; i < m; i ++) {
        pp[i] = new int [n]; //n为变量
}
```



动态分配的二维数组

pp[i][j]的访问过程：

1. 读取 pp 的值，再通过 *(pp+i) 得到 pp[i] 的值
2. 通过 *(*(pp+i)+j) 得到 pp[i][j]的值



动态分配的二维数组

二维数组回收

```
int m, n;
cin>>m>>n;
int **pp = new int* [m]; //m为变量
for(int i = 0; i < m; i ++) {
        pp[i] = new int [n]; //n为变量
}
```

回收

```
for(int i = 0; i < m; i ++) {
        delete [] pp[i];
}
delete []pp;
```

动态分配的二维数组

动态生成m行n列的二维数组，n为常量

```
int m;
cin>>m;
constant int n = 2;//n必须为常量,不能由用户输入
int (*pp)[n] = new int [m][n];//m可以为变量
```

回收

```
delete []pp;
```

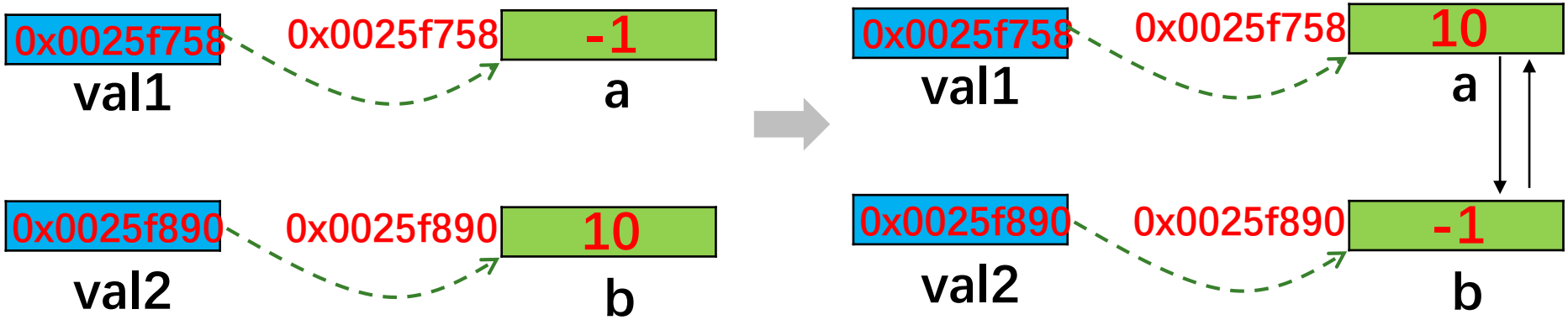
指针与函数

指针作为函数形参：变量地址作为实参

&a　　&b

```
int main() {
        int a = -1, b = 10;
        swap(&a, &b);
        return 0;
}
```

```
void swap (int *val1, int *val2)
{
        int tmp = *val1;
        *val1 = *val2;
        *val2 = tmp;
}
```



END