高级语言C++程序设计
Lecture 7 指针

南开大学 计算机学院

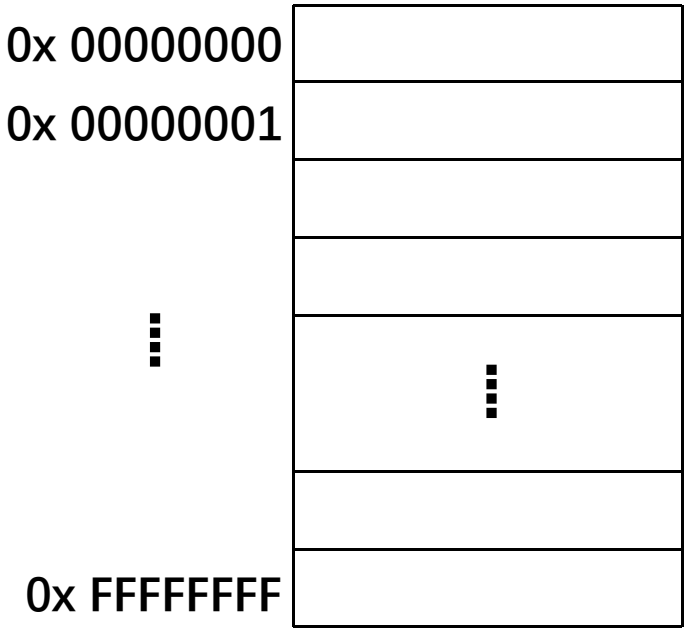
2021

指针的由来

计算机内存

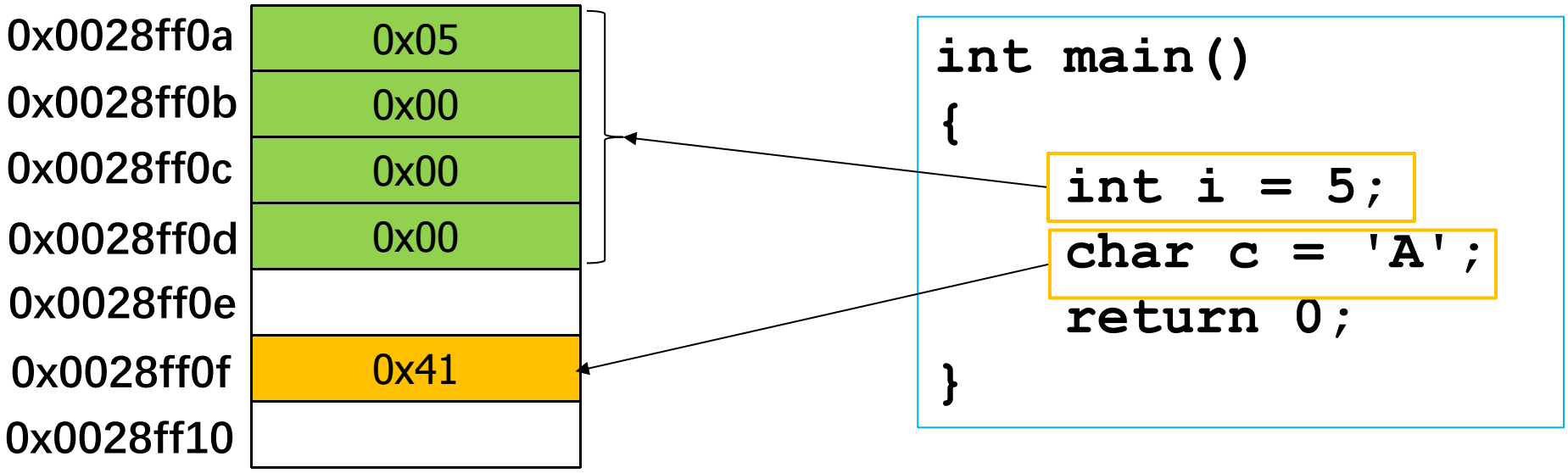
内存：计算机的存储空间，用于程序运行时保存数据 (程序指令、变量等)

- ✓ 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 0～2^{32}-1



内存地址

变量在内存中的地址

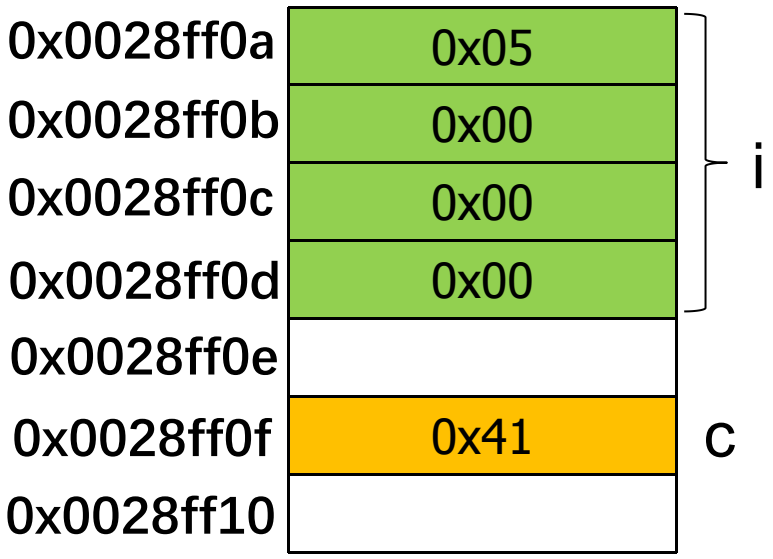


变量 i 在内存中的（起始）地址为 0x0028ff0a

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>



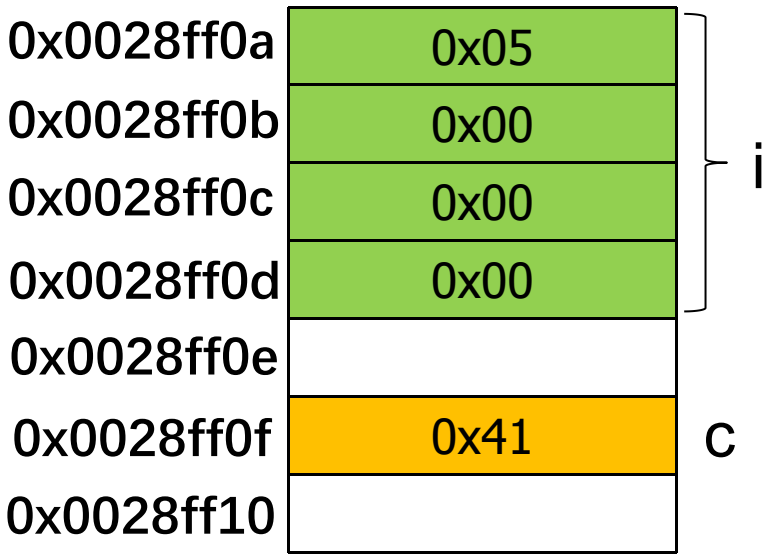

```
int main()
{
    int i = 5;
    char c = 'A';
    cout<<&i<<endl;
    cout<<(void*)&c<<endl;
    return 0;
}
```

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？

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

内存地址

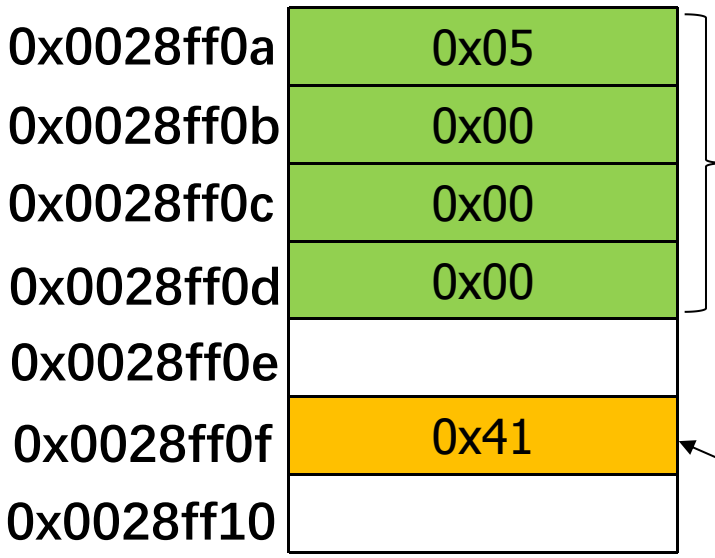
用什么**数据类型**表示内存地址？

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

方案一：定义新数据类型 address，占32 bits

内存地址

用什么数据类型表示内存地址？




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

执行*p操作（由地址取变量）需要两个信息:(1) p的值（代表变量起始位置); (2)变量的类型

address类型不包含第(2)个信息，因此方案一行不通！

内存地址

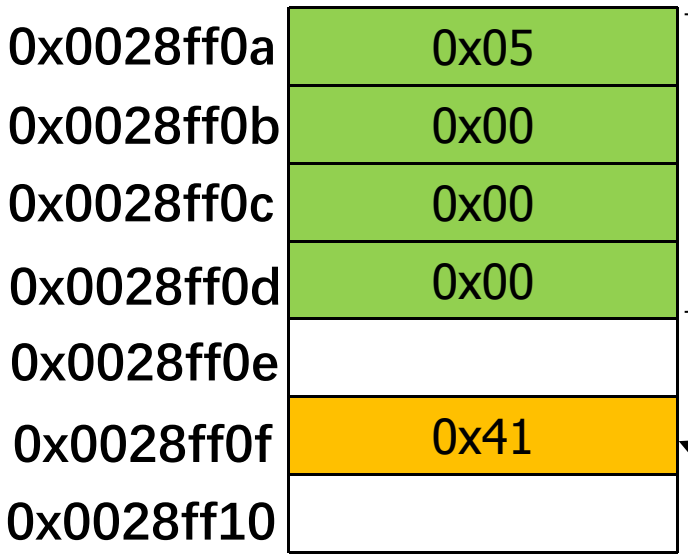
用什么**数据类型**表示内存地址？

方案二：
<变量类型> + address

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

内存地址

用什么数据类型表示内存地址？




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

p的类型为int address，表示p是int类型变量的地址，因此，执行*p的时候解释成int 类型的变量

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

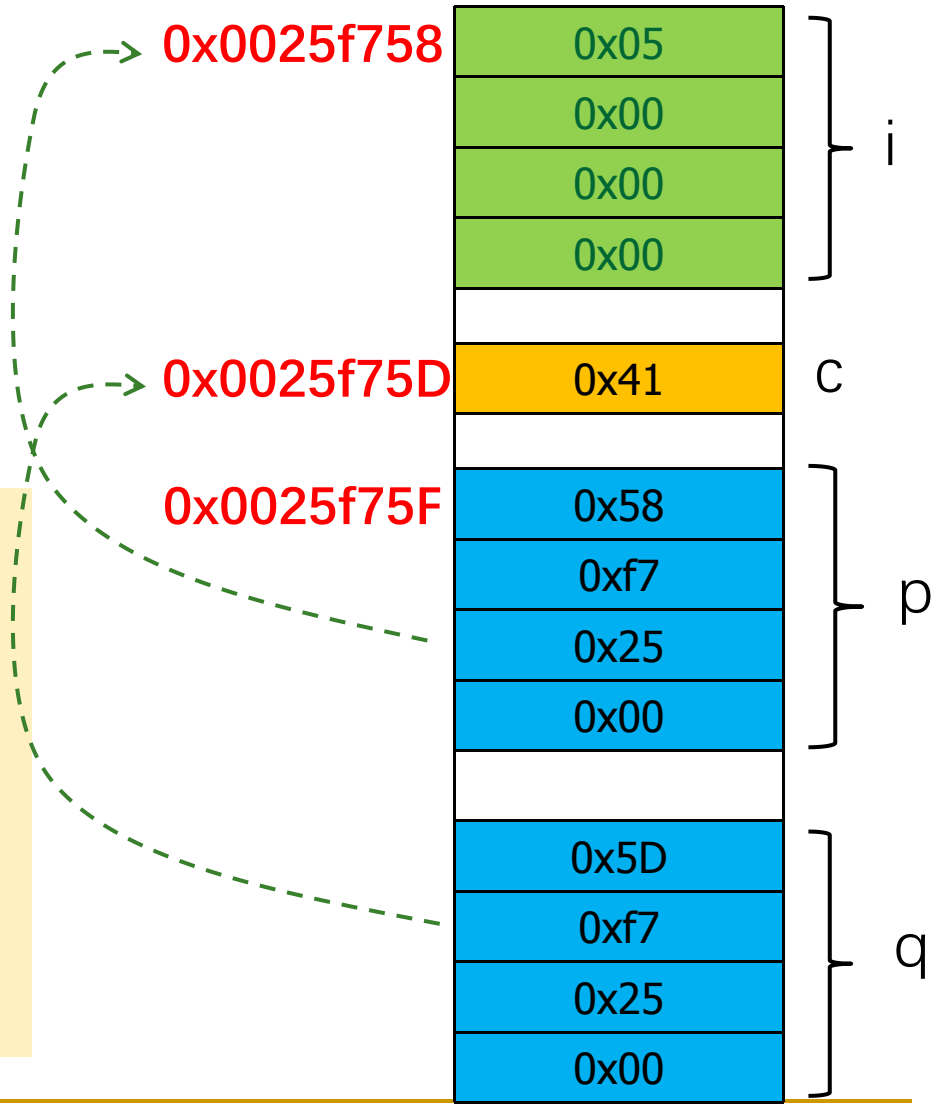
指针

```
int i = 5;
int *p = &i;
char c = 'A';
char *q = &c;
```

p和q都是指针变量：

p的类型是 int *（int 型数据的地址）

q的类型是 char *（char 型数据的地址）



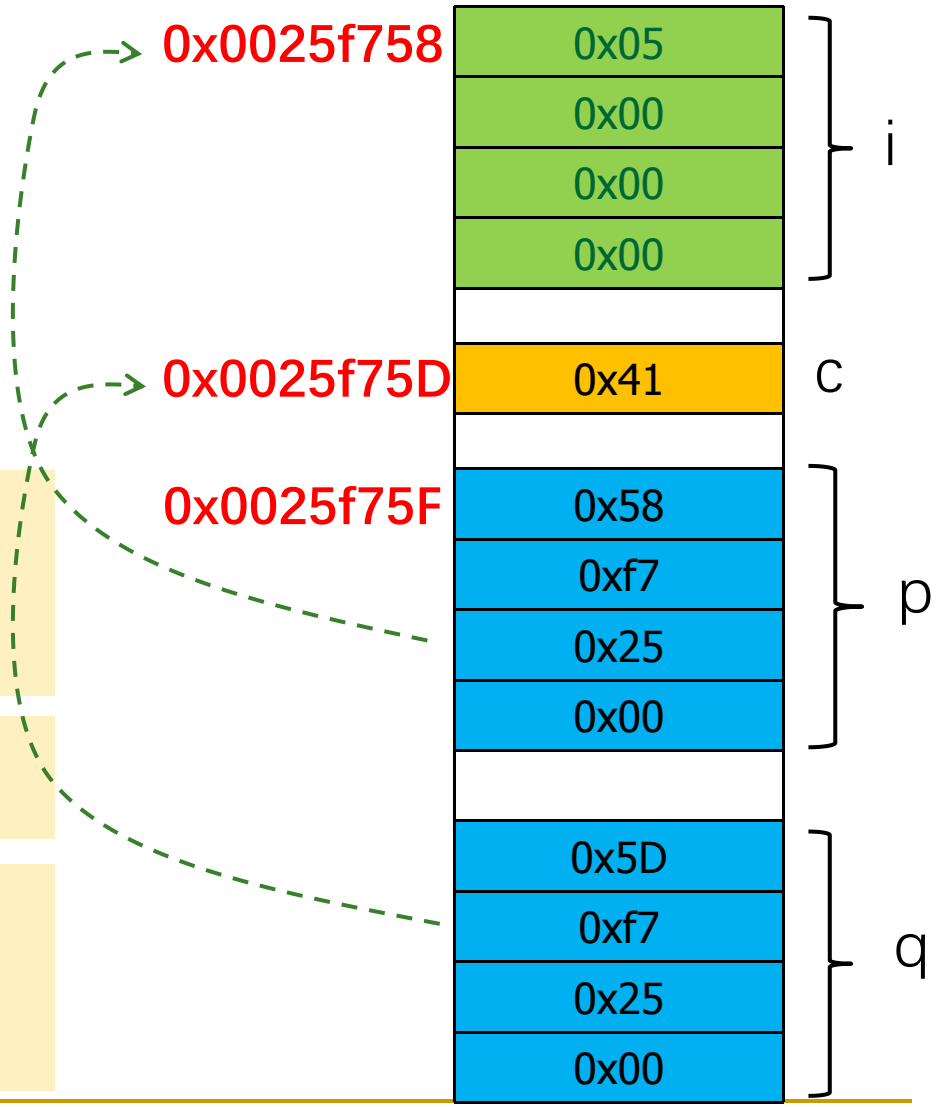
指针

```
int i = 5;
int *p = &i;
char c = 'A';
char *q = &c;
```

p存储变量i的地址，
q存储变量c的地址

p和q各占4个字节

与其他类型的变量一样，p和q也存在内存中



指针

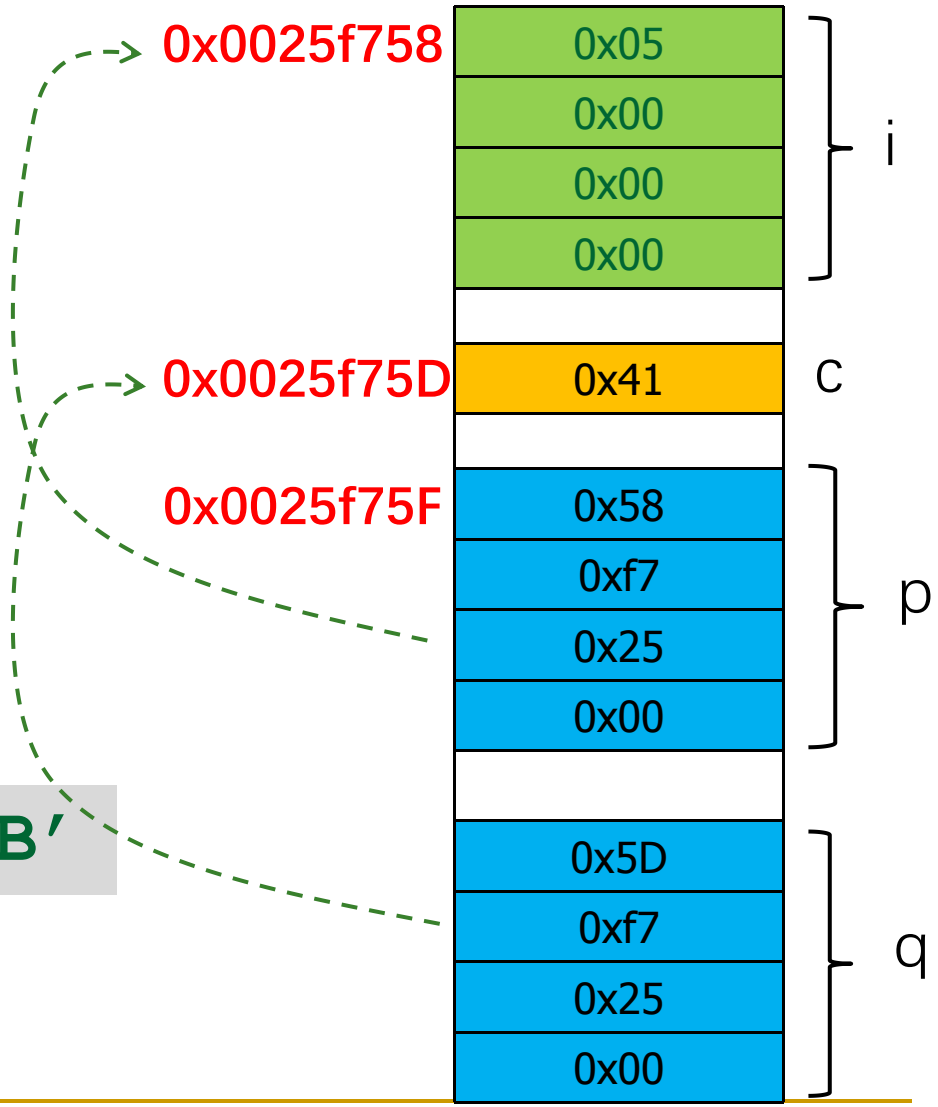
```
int i = 5;
int *p = &i;
char c = 'A';
char *q = &c;
```

```
cout<<*p<<endl;
```

通过p可以访问变量i，p称为i的指针

```
*q = 'B'; //将c变为'B'
```

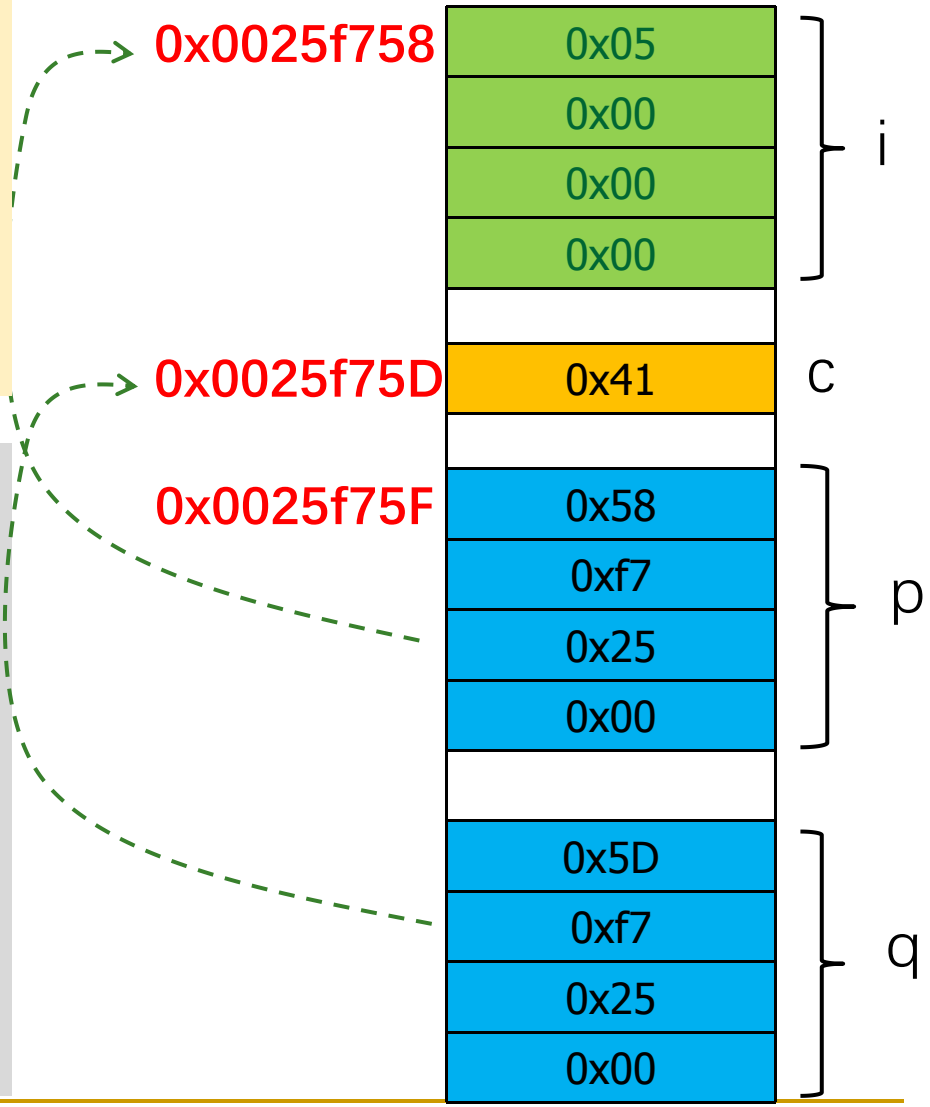
通过q可以访问变量c，q称为c的指针



指针

在C++ 11中，可以用auto 关键字来定义指针，编译器会自己分析指针的具体类型...

```
int i = 5;
auto *p = &i;
char c = 'A';
auto *q = &c;
cout<<*p<<endl;
*q = 'B';
cout<<*q<<endl;
```



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
```

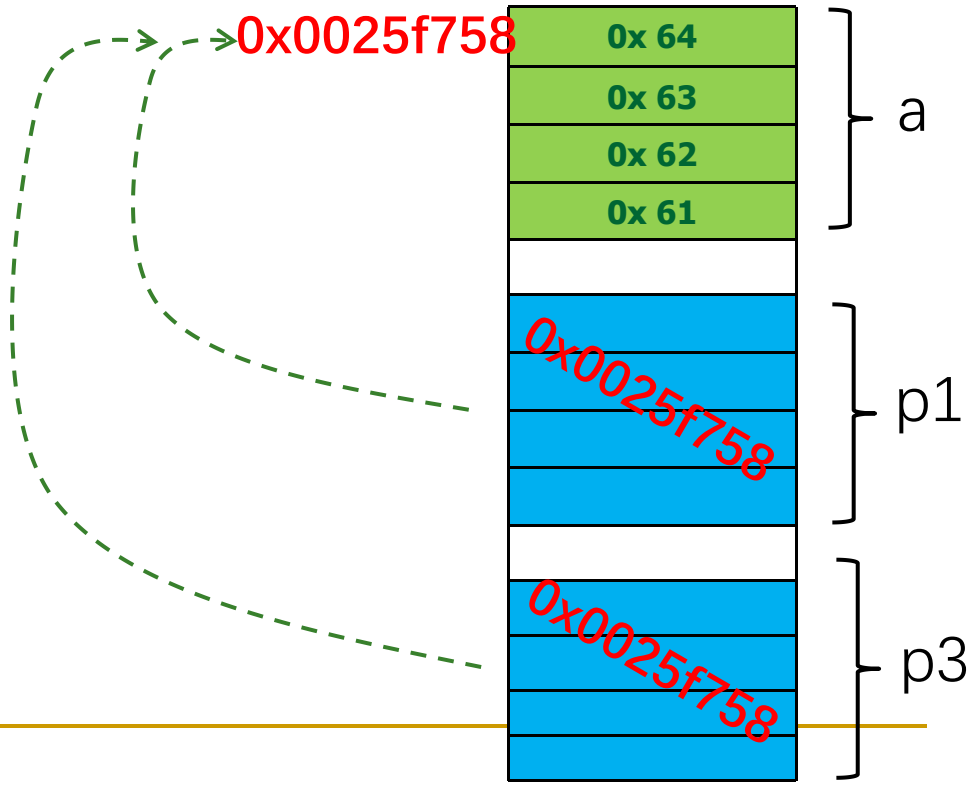
指针初始化

强制类型转换引起的问题

```
int a = 0x61626364;
int *p1 = &a; //ok
char *p3 = (char*)&a; //强制类型转换
```

```
cout<<*p1<<*p3;
```

p3是char *类型，*p3表示一个字符，即0x64（字符'd'）

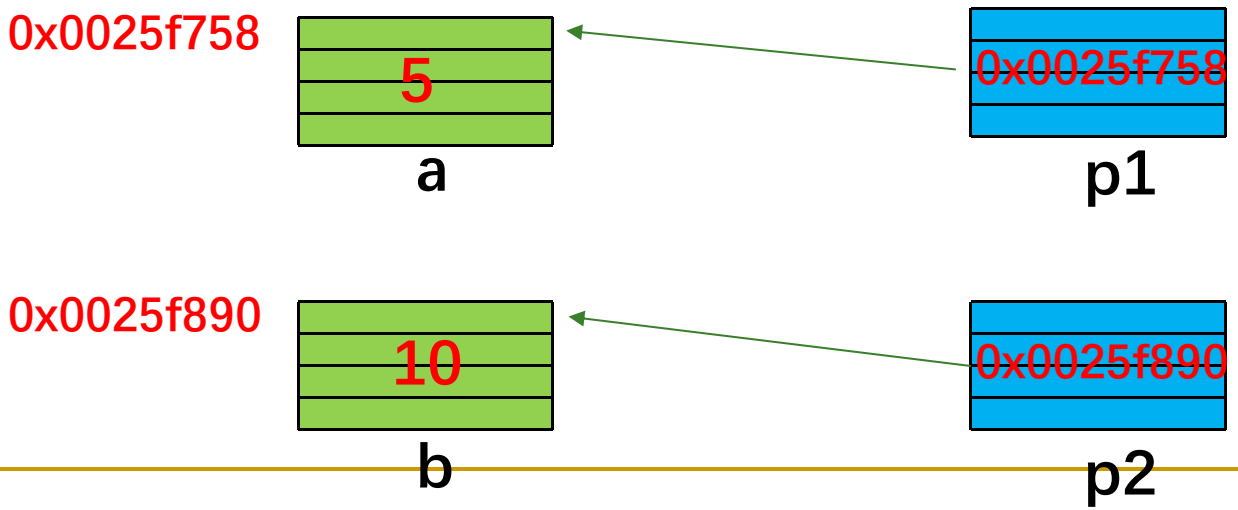


指针基本操作

通过间接访问运算符访问指针所指向的变量

* <指针表达式>

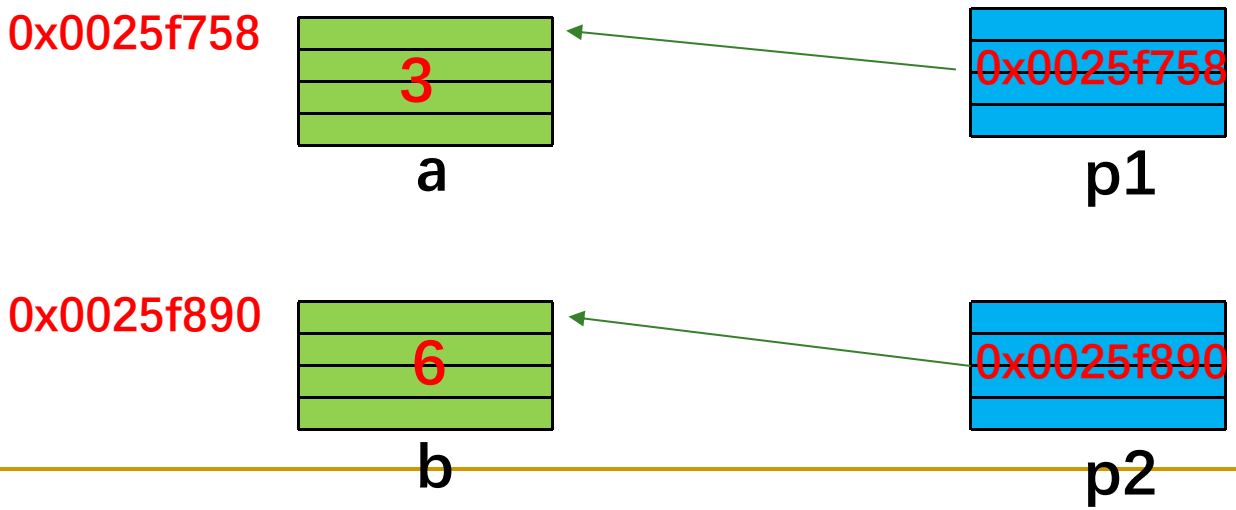
```
int a = 5, b = 10;
int  *p1, *p2;
p1 = &a; p2 = &b;
cout<<a<<b<<*p1<<*p2;
```



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;
```

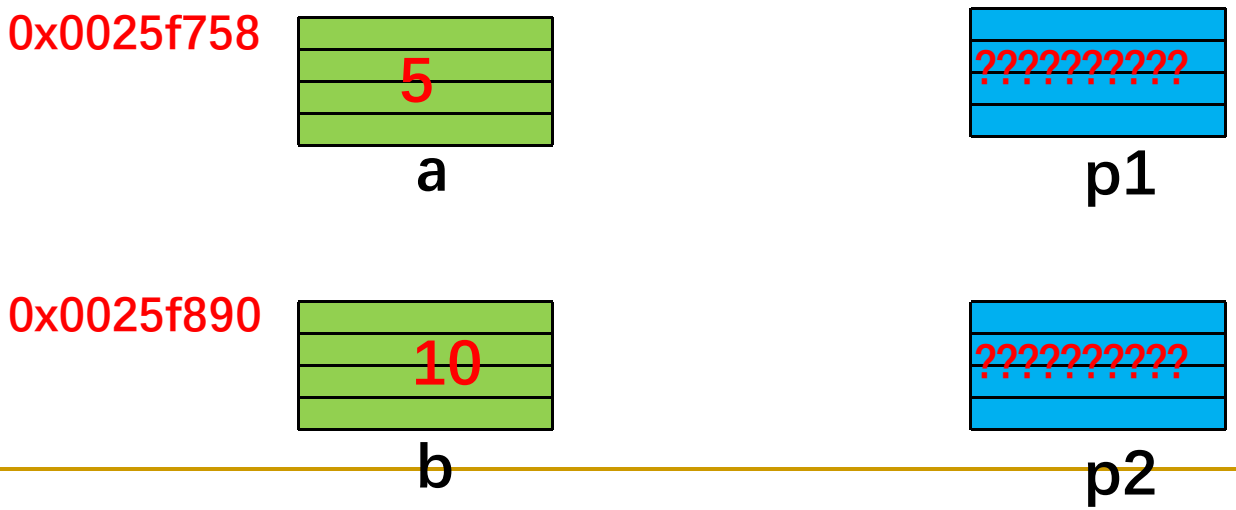
*p1对应的是变量a，因此可以作为左值



指针基本操作

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

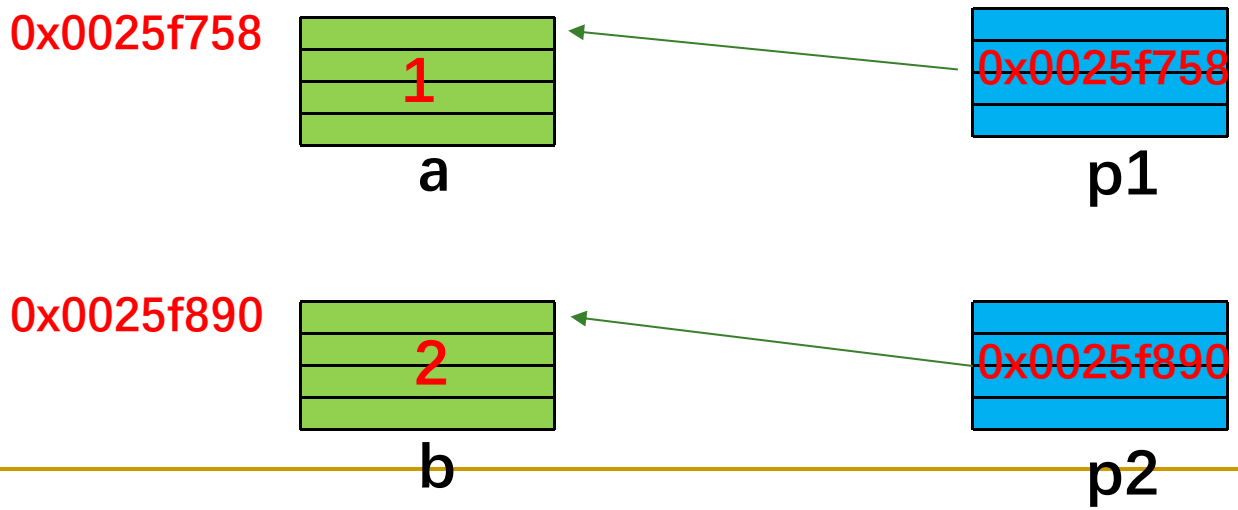
Fatal Error! 作为左值之前必须初始化！否则指向未知区域



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;
```

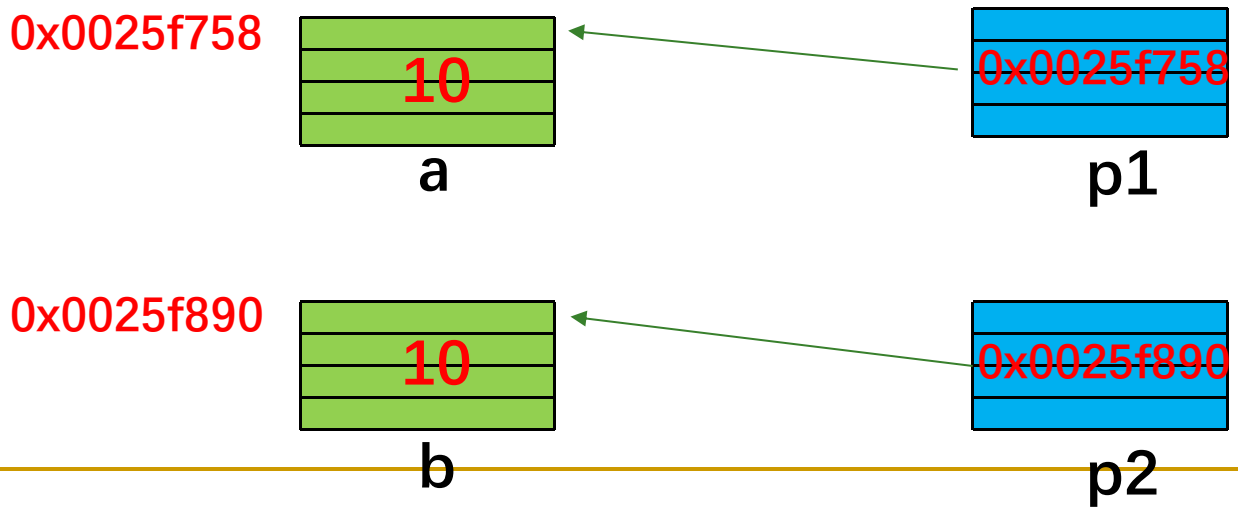
变量改动与指针无关



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;
```

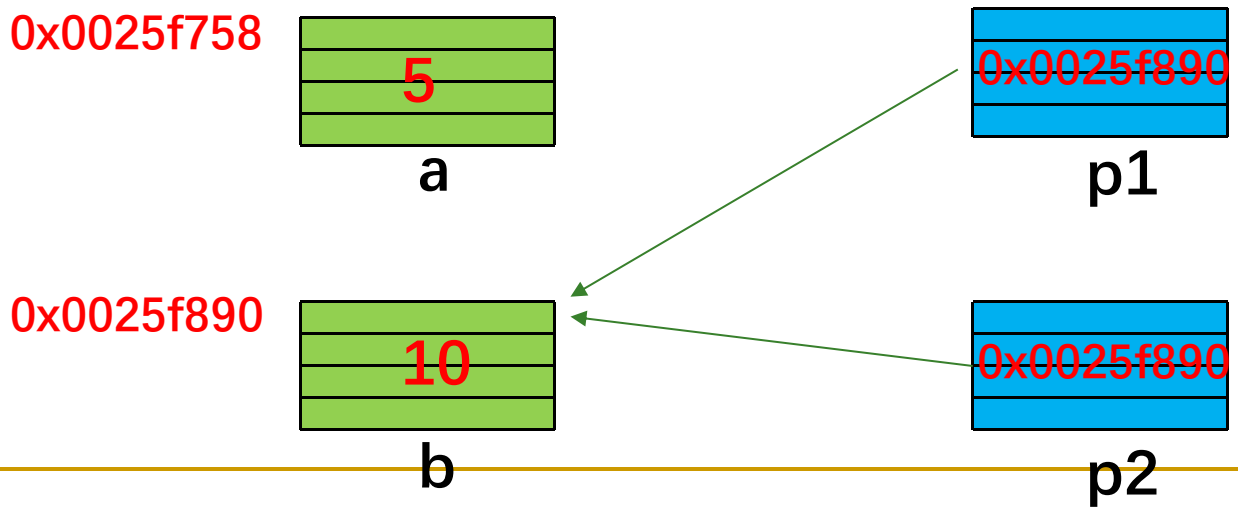
*p1作为左值，*p2作为右值



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

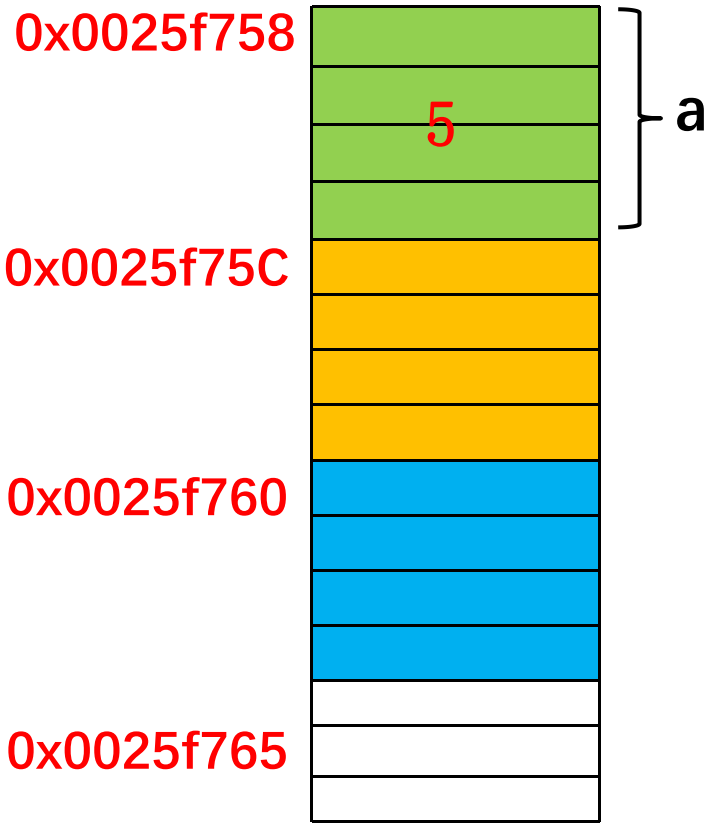


指针运算

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)

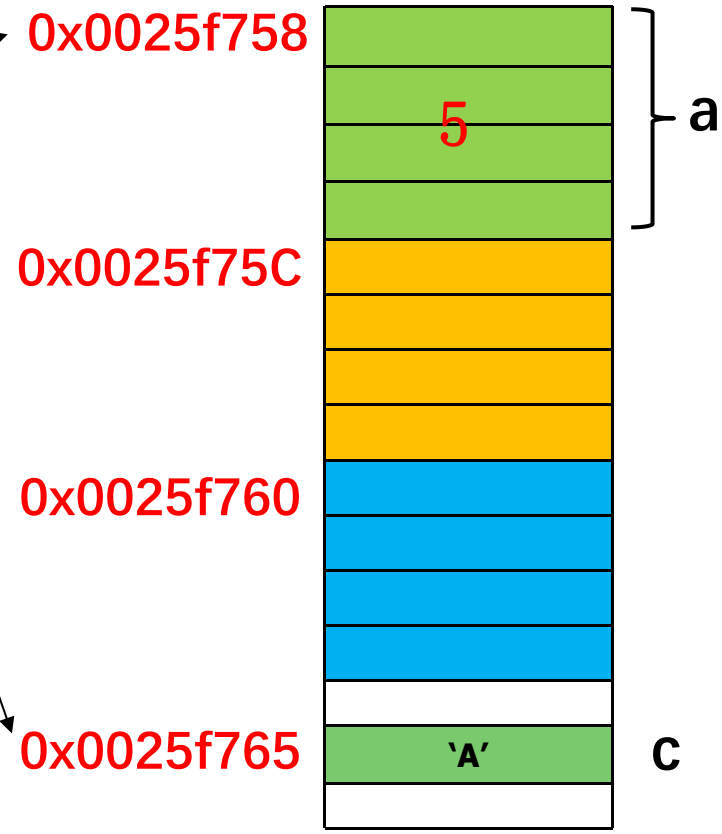
```
p ++; //p+1*4
cout<<*(p-3);
```

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

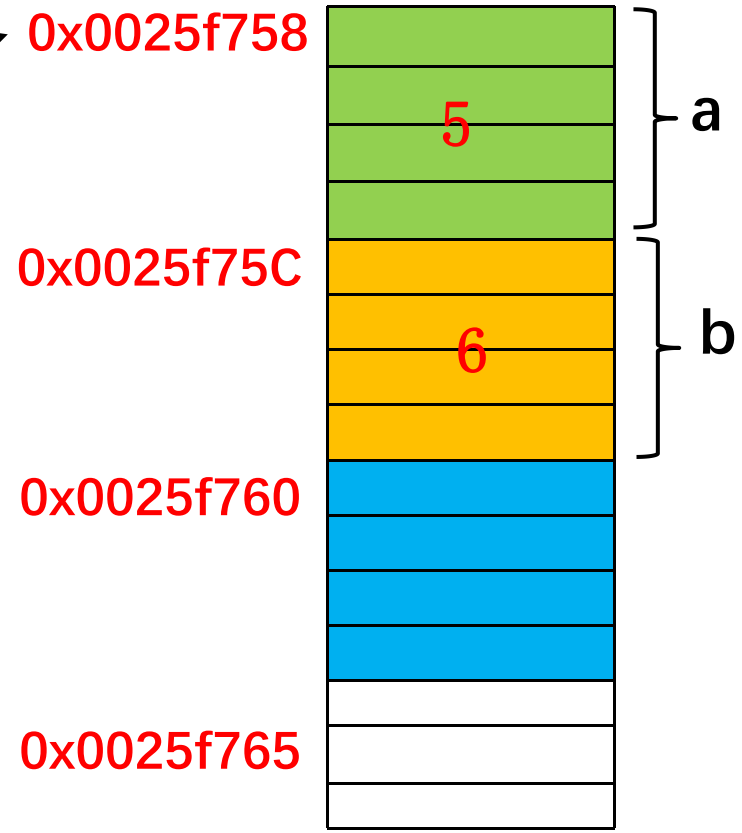
```
int a = 5;
int  *p = &a;
p = p + 2; //p+2*4
char c = 'A';
char *p1 = &c;
p1 = p1 - 5; //p1-5*1
```



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;
```



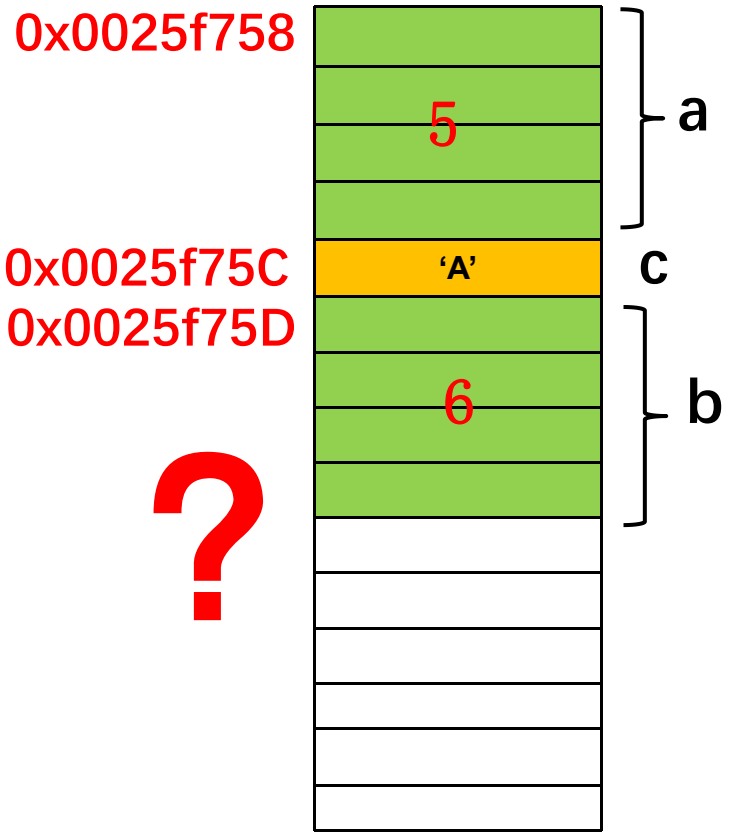
同类型指针相减等于两个指针之间所指类型数据的个数

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;
char c = 'A';
int b = 6;
int *p1 = &a;
int *p2 = &b;
cout<<p2-p1;
```



两个int变量之间有一个char变量，内存如何分配？

如果连续存储，p2-p1不是4字节的倍数！

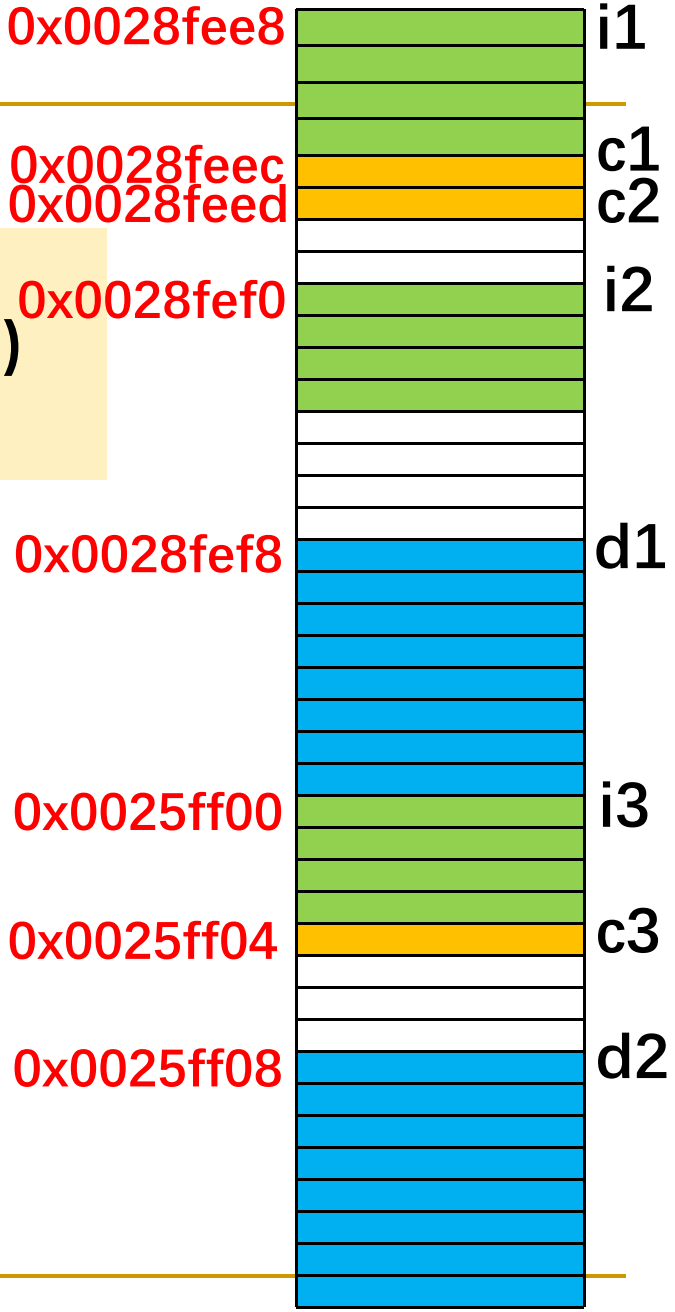
内存对齐

变量地址是变量大小的倍数

int变量地址是4的倍数（4字节对齐）
double变量的地址是8的倍数（8字节对齐）
char变量的地址是1的倍数（1字节对齐）

```
struct test{
    int i1;
    char c1;
    char c2;
    int i2;
    double d1;
    int i3;
    char c3;
    double d2;
};
```

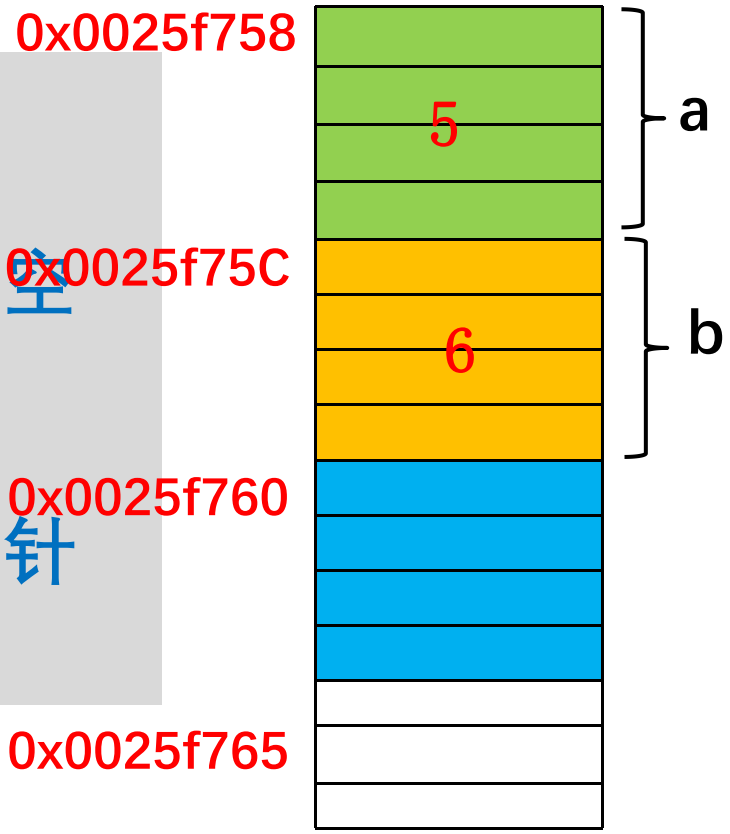
思考：以什么顺序定义这些变量最省空间？



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++;
```



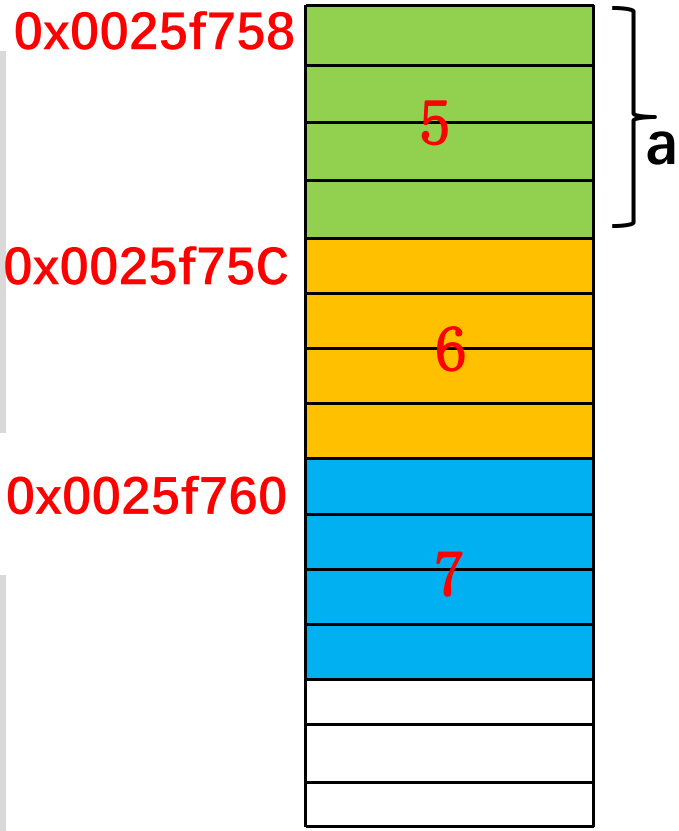
只有相同类型指针才能够进行比较！

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];
```

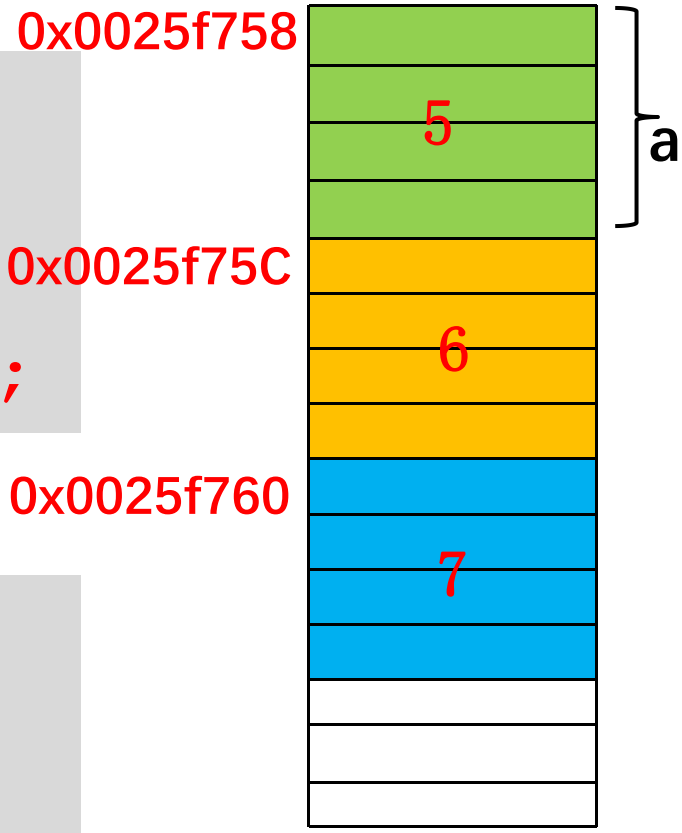
```
p[0] == *p
p[1] == *(p+1)
p[2] == *(p+2)
```



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];
```

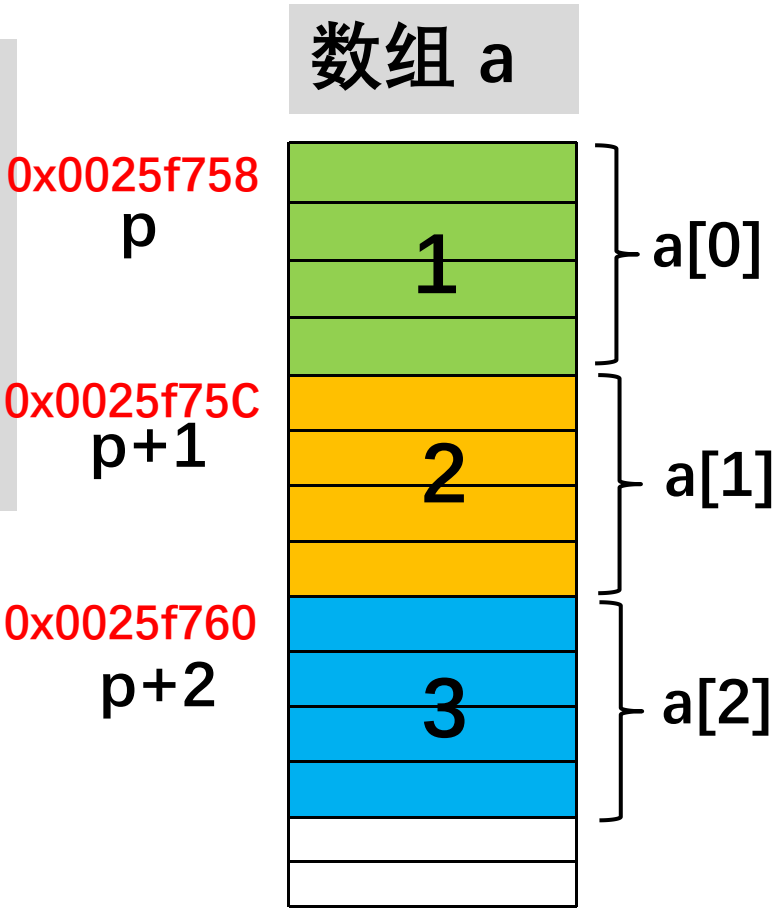



```
&p[0] == p
&p[1] == (p+1)
&p[2] == (p+2)
```

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];
```

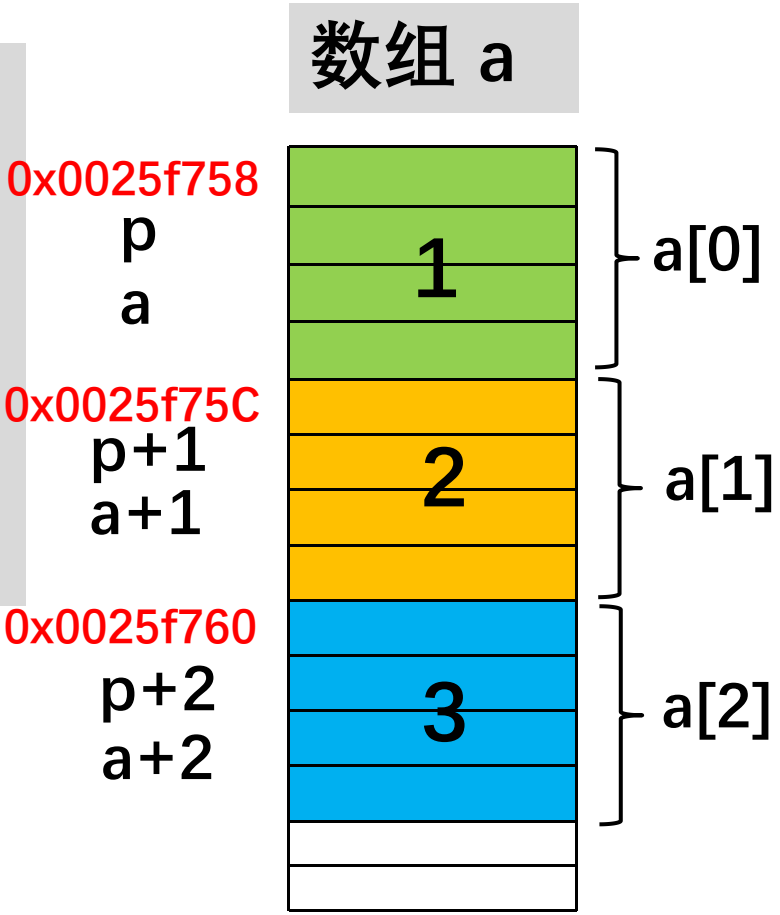


p是指向数组首元素的指针，
p实现与数组名a相同的功能

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]

指针与数组

数组名是指针常量，不能改变，不能作为左值

指针是变量，可以修改，可以作为左值

```
int a[3], b[4] ;
int *p = &a[0] ; //ok
p = b; //ok, p是变量可以作为左值
a = p; //error, a是指针常量，不能作为左值
b = p; //error, b是指针常量，不能作为左值
p++; //ok
a++; //error, a是指针常量，不能修改
```

指针与数组

数组名的双重身份

```
int a[3] = {1, 2, 3};
cout<<*a; //a代表首元素的指针
cout<<a[0]; //a代表首元素的指针
cout<<sizeof(a); //a代表整个数组，输出12
int *p;
p = a; //a代表首元素指针
```

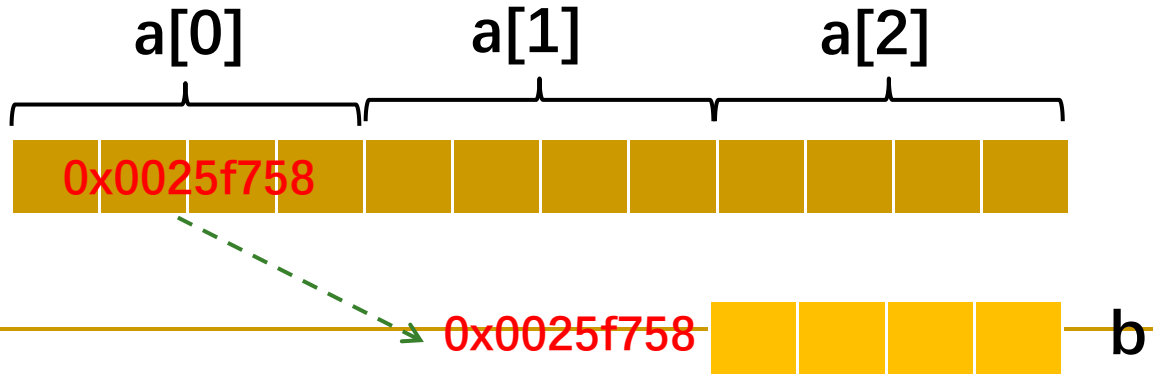
指针数组

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

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[0]); //ok?
```

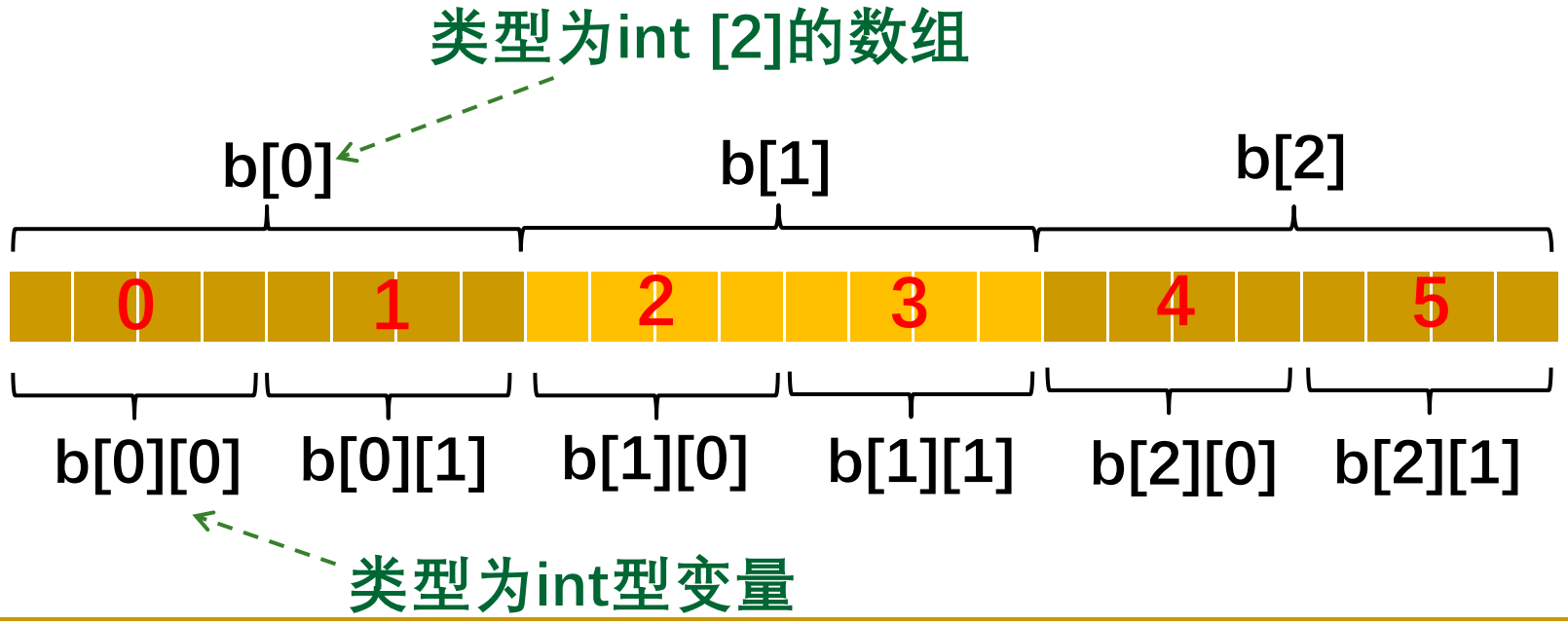
错误，&b[0]是数组首元素的地址，是一个数值，不占用任何内存空间，不能再取地址！

指针与二维数组

二维数组：元素为一维数组的数组

```
int b[3][2] = {{0,1}, {2,3}, {4,5}};
```

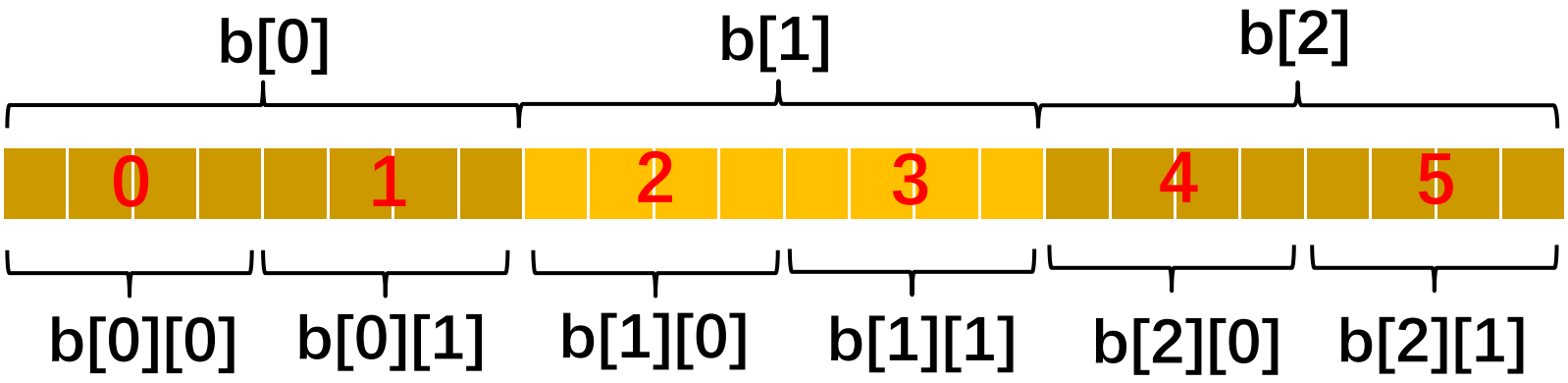
分析：b[3]表明b是一个数组，有3个元素，剩余部分（即int [2]）为元素类型（大小为2的整型数组）



指针与二维数组

二维数组与指针

```
int b[3][2] = {{0,1}, {2,3}, {4,5}};
```



数组名是指向数组首元素的指针

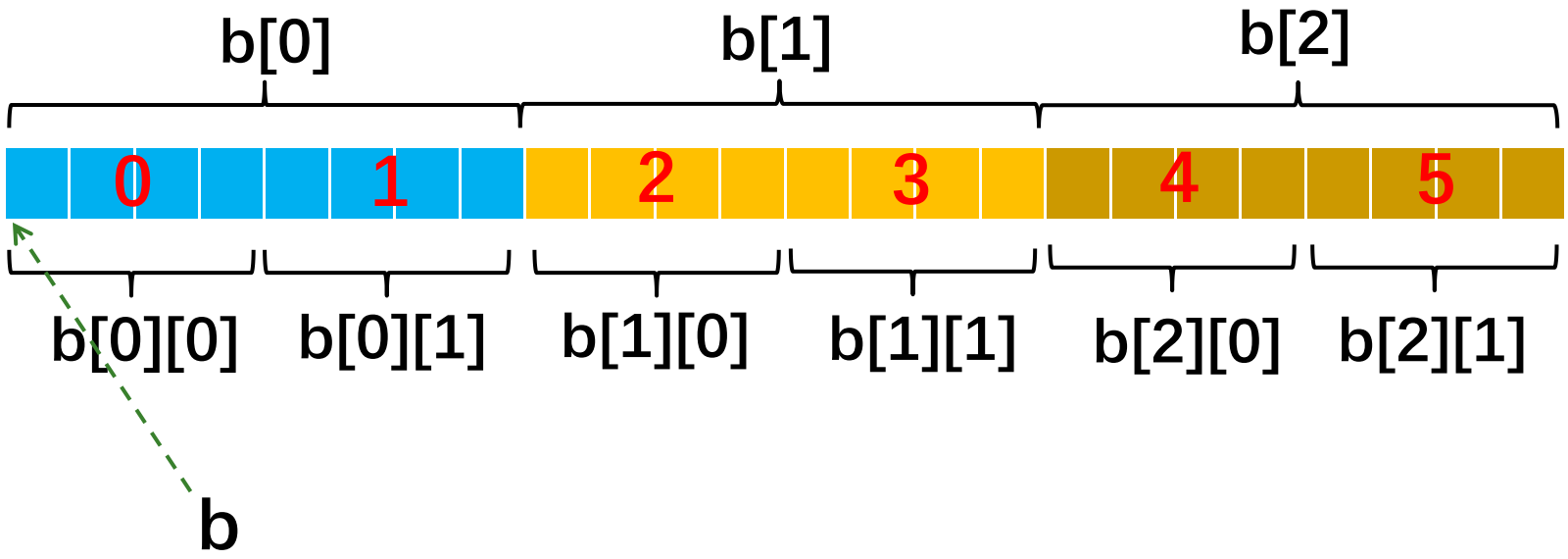
b 是指向首元素b[0]的指针，即 b的值为&b[0]

b[0]的类型为int [2]型，因此b是指向 int [2]型数组的指针

指针与二维数组

二维数组与指针

```
int b[3][2] = {{0,1}, {2,3}, {4,5}};
```

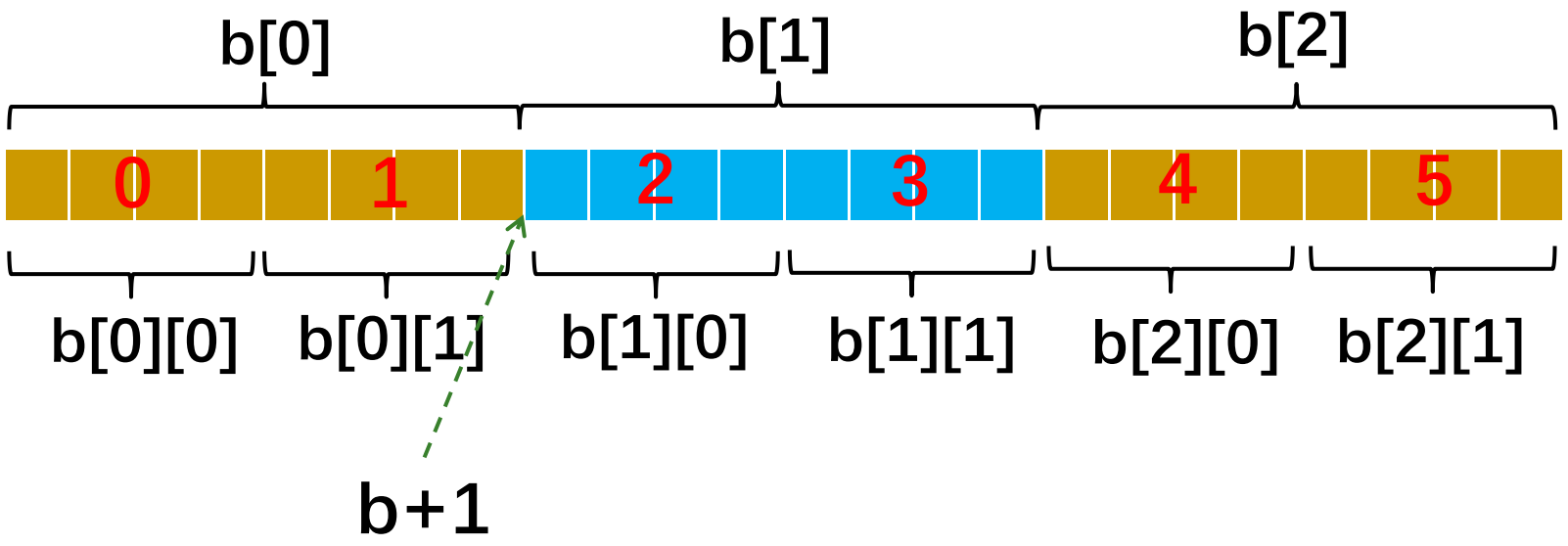


b == &b[0]，指向int [2]型数组的指针

指针与二维数组

二维数组与指针

```
int b[3][2] = {{0,1}, {2,3}, {4,5}};
```

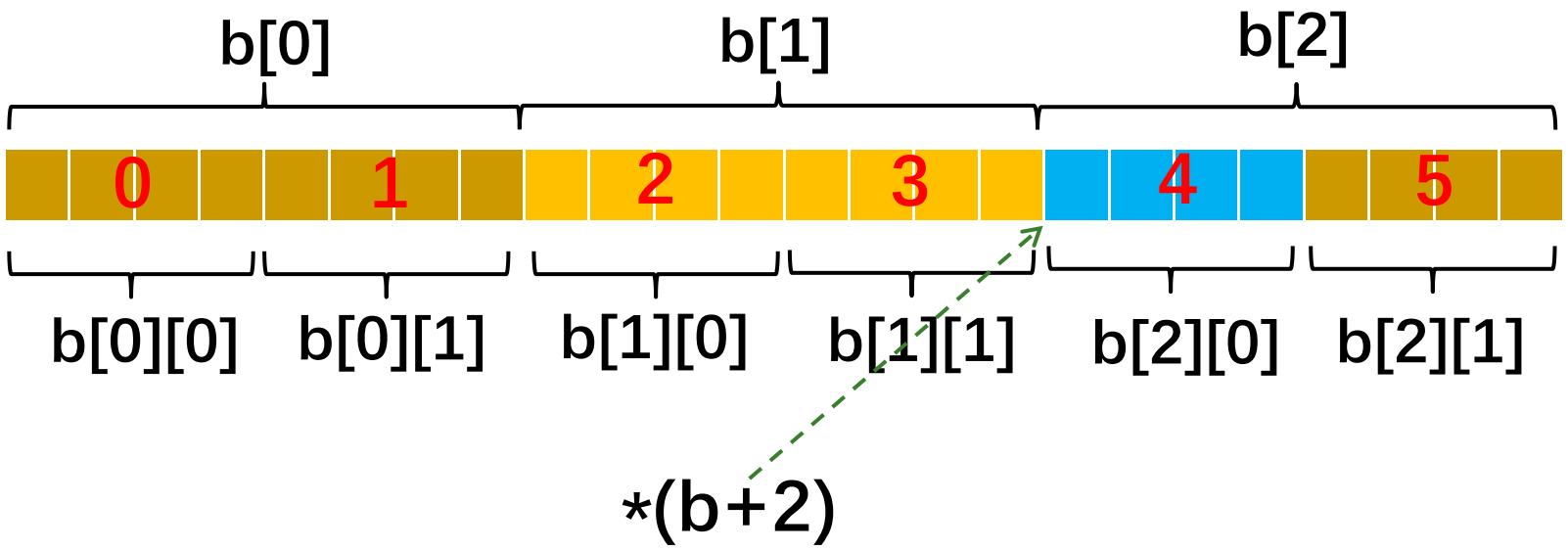


b+1 的值为 b + sizeof (int [2])，等价于 &b[1]

指针与二维数组

二维数组与指针

```
int b[3][2] = {{0,1}, {2,3}, {4,5}};
```

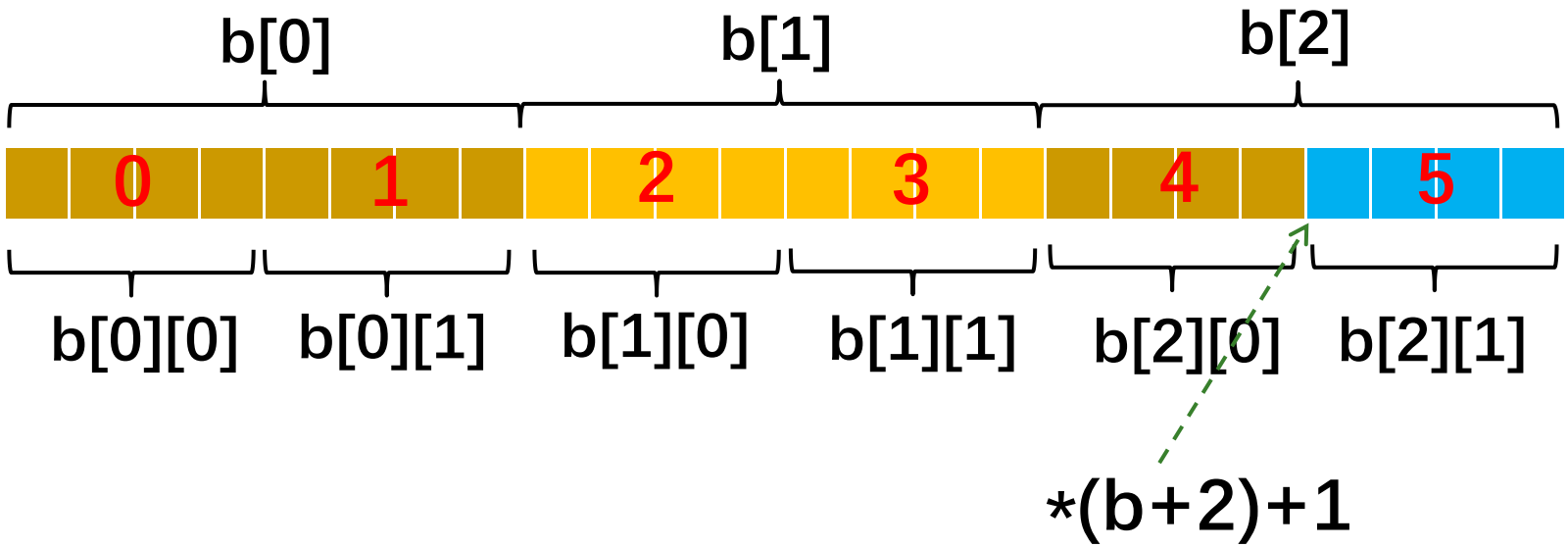


*(b+2) == b[2] == &b[2][0]，即指向b[2][0]的指针

指针与二维数组

二维数组与指针

```
int b[3][2] = {{0,1}, {2,3}, {4,5}};
```

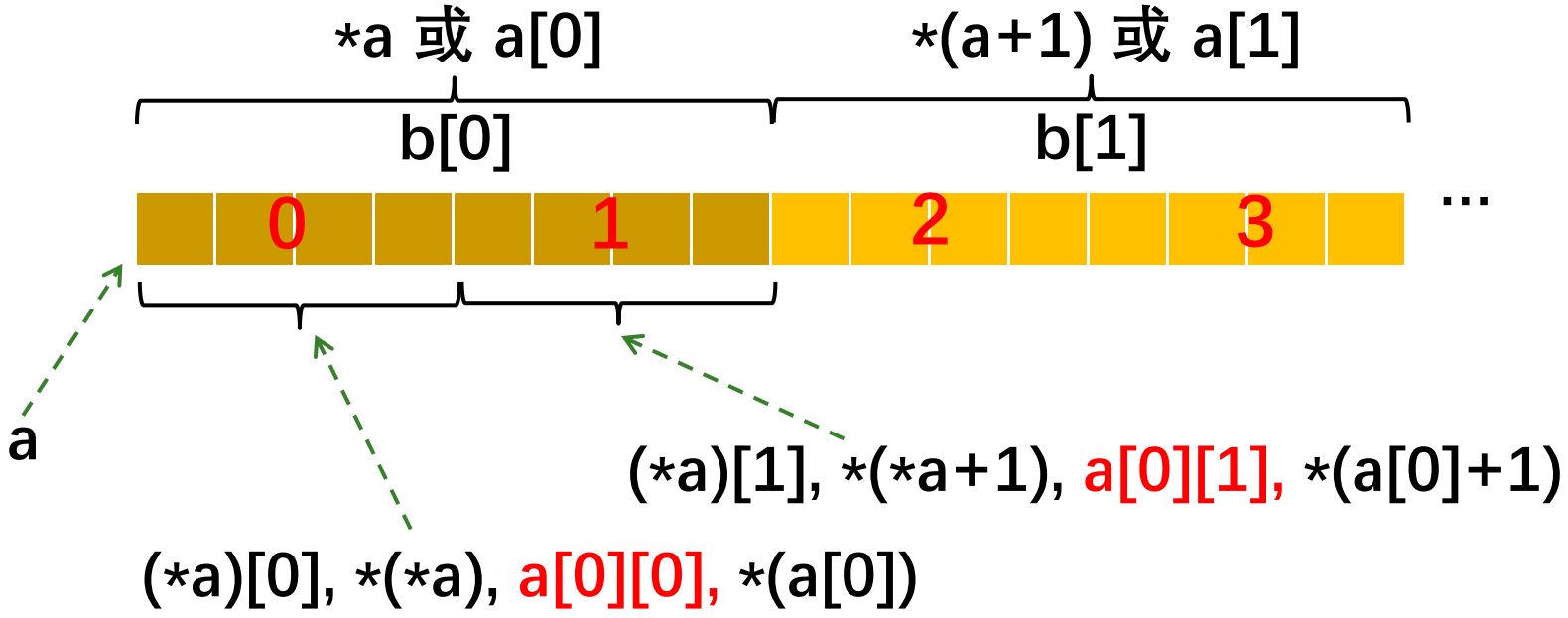


*(b+2)+1 == b[2] + 1 == &b[2][1]，即b[2][1]的地址

指针与二维数组

二维数组与指针

```
int b[3][2] = {{0,1}, {2,3}, {4,5}};
int (*a) [2] = b ;
```



a相当于二维数组名b！

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]

练习

```
int A[3][4] = {1,2,3,4,5,6,7,8,9,10};
int (*pa)[4] = A;
cout<<*pa[1]<<endl;
cout<<(*pa)[1]<<endl;
cout<<*(pa+1)[1]<<endl;
```

输出结果：
5
2
9

- A是数组名，指向首元素（int [4]类型）的指针，pa是指针，指向 int [4]类型
- []优先级高于*，*pa[1]等价于*(p[1])，等价于pa[1][0]，等价于A[1][0]
- (*pa)[1]等价于(pa[0])[1],等价于pa[0][1]
- *(pa+1)[1]等价于*(*((pa+1)+1))，等价于**(pa+2)，等价于*pa[2]，等价于pa[2][0]

指针与二维数组

例子

```
int *p1;
int b[2][3];
int c[3][2];
int (*p2)[3]; //指向int [3]型数组的指针
```

```
p1 = &b[0][0]; //ok?
```

```
p1 = b[0]; //ok?
```

```
p1 = b; //ok?
```

```
p2 = b; //ok?
```

```
p2 = c; //ok?
```

练习

```
int A[3][4]={0,2,4,6,8,10,12,14,16,18,20,22};
int (*pa)[4];
pa = A;
cout<<A<<" "<<A+1<<endl;
cout<<&A<<" "<<&A+1<<endl;
cout<<*A<<" "<<*A+1<<endl;
cout<<A[0]<<" "<<A[0]+1<<endl;
cout<<&A[0]<<" "<<&A[0]+1<<endl;
cout<<*A[0]<<" "<<*A[0]+1<<endl;
cout<<A[0][0]<<" "<<A[0][0]+1<<endl;
cout<<&A[0][0]<<" "<<&A[0][0]+1<<endl;
```

```
0x28fedc 0x28feec
0x28fedc 0x28ff0c
0x28fedc 0x28fee0
0x28fedc 0x28fee0
0x28fedc 0x28feec
0 1
0 1
0x28fedc 0x28fee0
```

分析

设A为整型二维数组，将A赋值给一维数组指针pa

表达式	值	含义	表达式+1	等价的pa
A	&A[0]	数组A的首元素地址	下一个元素的地址，即A+1(&A[1])，相差16B	pa
&A	数组A的地址	指向数组A的指针，其值为数组A的地址	下一个存储单元，即&A+1，相差48B	&(*pa)
*A	A[0]	数组A[0]的首元素地址	下一个元素的地址，即*A+1(&A[0][1])，相差4B	*pa
A[0]	同*A，是数组A[0]名字，也是A[0]的首元素地址，即&A[0][0]			*(pa+0)
&A[0]	同A	指向数组A[0]的指针，其值为数组A[0]的地址	下一个存储单元，即&A[1]，相差16B	&(*(pa+0))
*A[0]	A[0][0]	数组A[0]的首元素值	数组A[0]的首元素值加1，即A[0][0]+1	*(*(pa+0)+0)
A[0][0]	同*A[0]			*(*(pa+0)+0)
&A[0][0]	A[0][0]的地址	数组元素A[0][0]的地址	下一个元素的地址，即&A[0][1]，相差4B	&(*(*(pa+0)+0))
*A[0][0]	×	语法错误	×	×

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
```

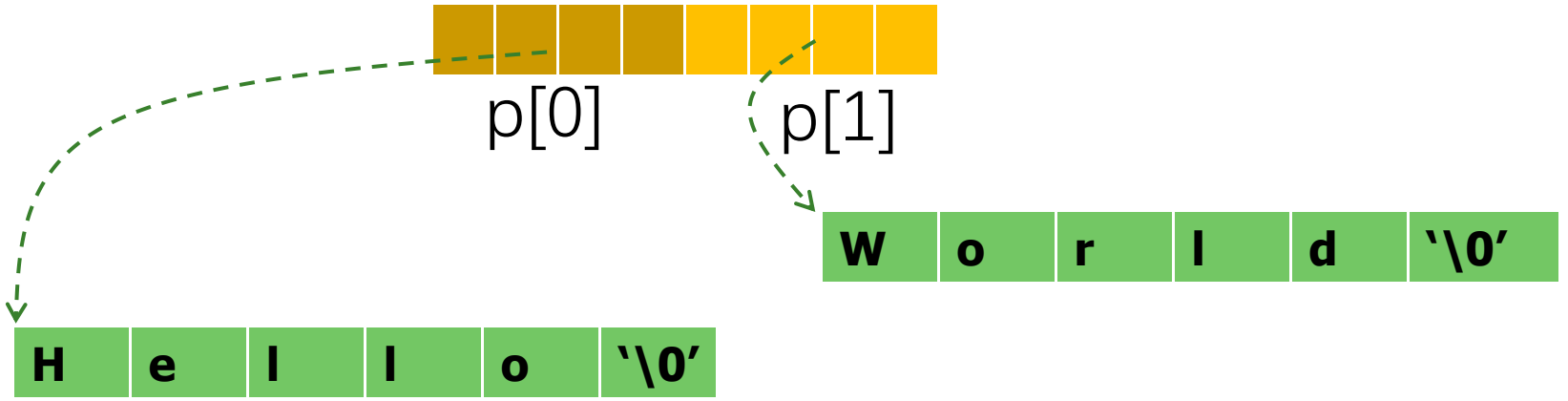
```
char c[] = "abc"
s = c// ok, s是指针变量，其值可以改
```

```
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！

指针与字符串

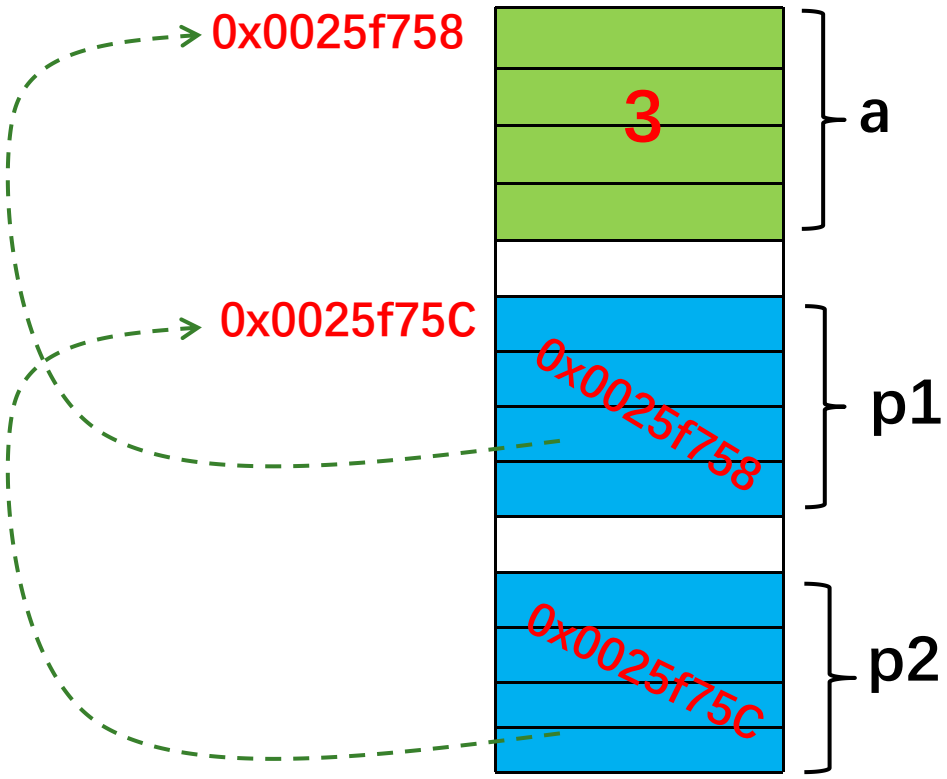
输出字符串地址（首地址）

```
#include<iostream>
using namespace std;
int main(){
   const char *string1="欢迎学习C++程序设计课程
   !"; //VS编译器要求const, GCC不需要
   cout<<"串值是:"<<string1<<"\n串地址是:"
   <<(void*)string1<<endl;
   return 0;
}
```

通常用输出数组名得到的是数组地址，但字符型数组（字符串）不同，输出的是数组内容。本例将字符指针强制转换为空指针输出字符串中第一个字符的地址。

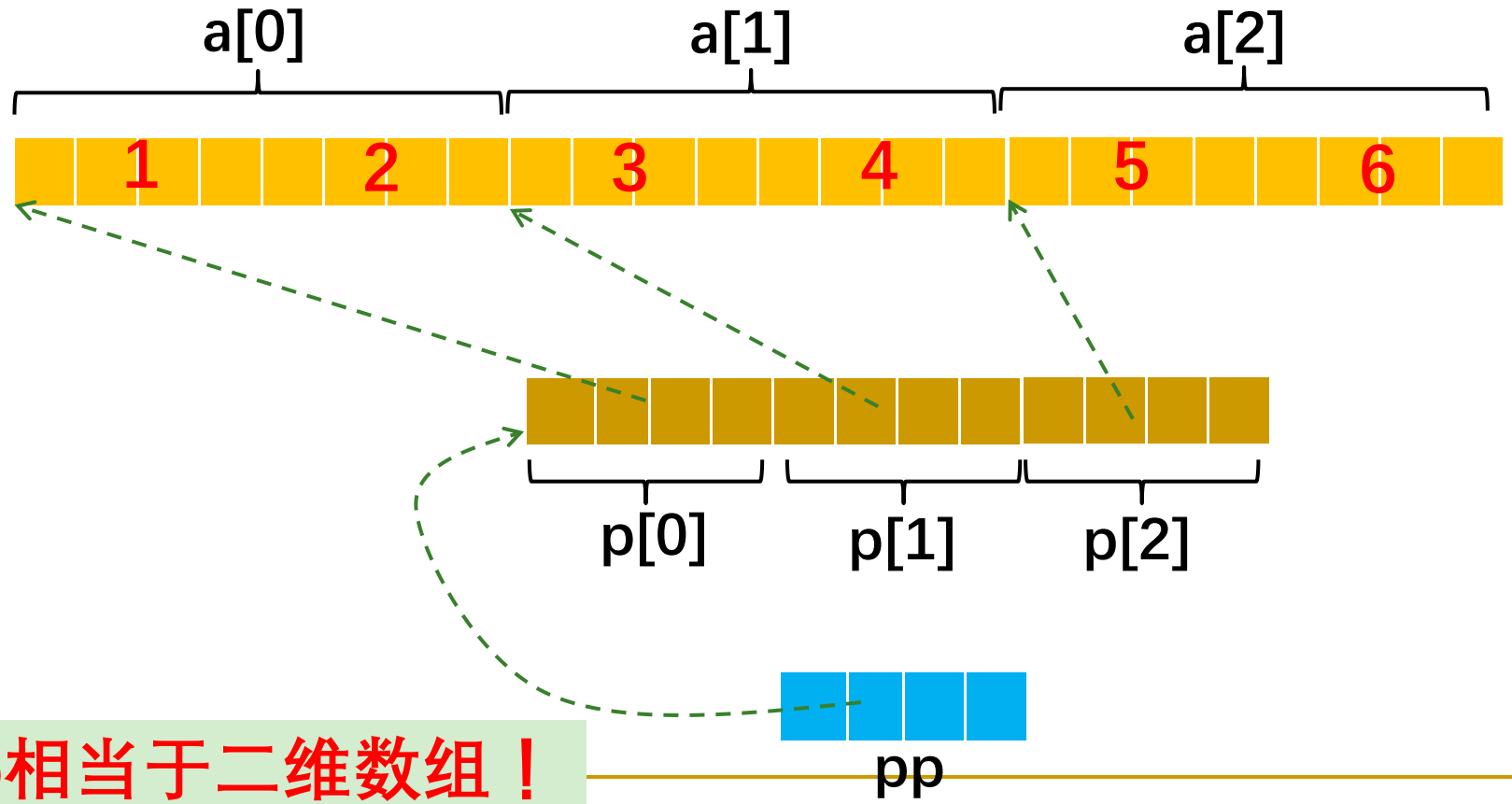
指针的指针

```
int a = 3;
int *p1 = &a;
int **p2 = &p1;
```



指针的指针

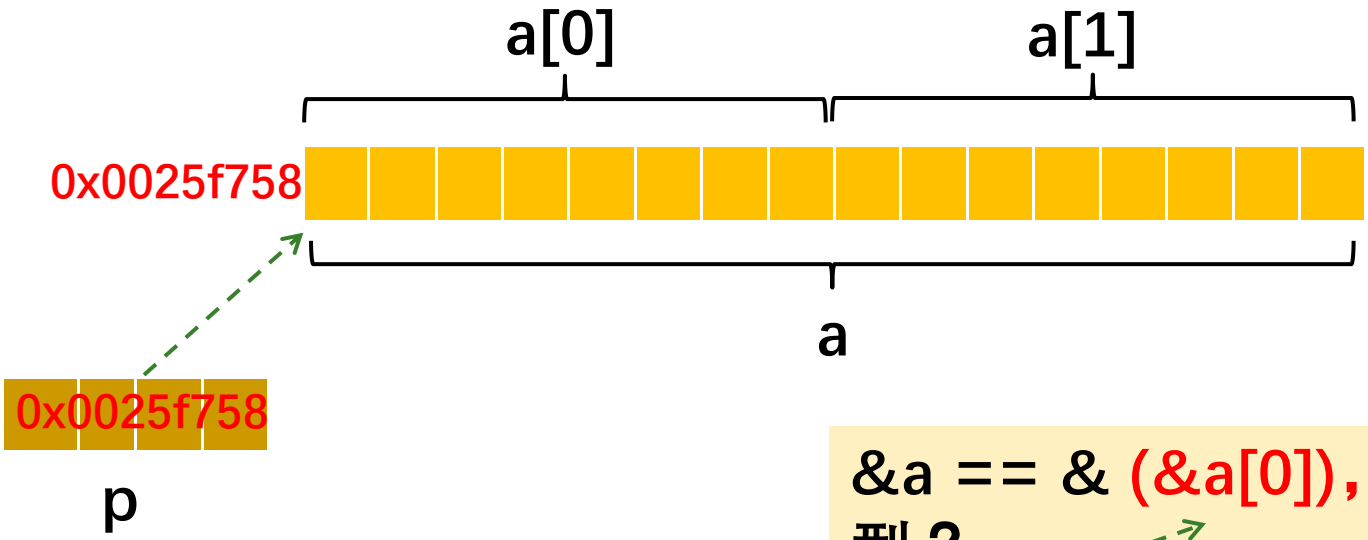
```
int a[][2] = {{1,2},{3,4},{5,6}};
int *p[3] = {a[0], a[1],a[2]};
int **pp = p;
```



pp相当于二维数组！

指针的指针

```
int a[2];
int (*p) [2] = &a;//ok?
int **pp = &a; //ok? &a == &(&a[0])?
```



a代表整个数组，是客观存在，取a的地址有意义，地址类型是数组指针

&a == & (&a[0])，是int **类型？

&a[0] 是数组首元素地址，是一个数字，没有存储空间，再取地址没有意义！

练习

```
#include<iostream>
using namespace std;
int main(){
  const char **p;
  const char *name[]={"BASIC","PASCAL","C++"};
  p = name + 2;
  cout<<p<<endl;//name数组第3个元素的地址
  cout<<*p<<endl;//name数组的第3个元素(char *类型)
  cout<<**p<<endl;
  //name数组的第3个元素指向的变量(char 类型)
  return 0;
}
```

运行结果:

```
0012FF78
C++
C
```

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

动态分配一个变量：

new <数据类型> (初始值)

```
int *p;
p = new int(3);
*p = 4;//变量赋值
```

new 返回变量的内存地址，p是指向变量的指针

动态分配的变量没有名字，需要通过指针p来访问

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

delete

first calls the appropriate destructor (for class types), and then calls a *deallocation function*.

先调用析构函数（如果是类对象），再回收动态分配的空间

delete []

first calls the appropriate destructors for each element in the array (if these are of a class type), and then calls an array deallocation function

先为每个数组元素调用析构函数（如果是类对象），再回收动态分配的空间

动态分配内存

```
class A{
    int x;
};
A *p;
p = new A(); //动态生成一个类A的对象
delete p; //先调用类A的析构函数，再回收A的空间

p = new A[10]; //动态生成10个类A的对象
delete p; //只为第一个类A的对象调用析构函数，然后释放动态空间（错误！）
delete []p; //为数组中每个类A的对象调用析构函数，再回收动态分配的空间
```

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

练习

```
int *p = new int [10];
for(int i = 0; i < 10; i ++) {
      *p = i;
       p ++;
}
delete []p;
```

动态空间的地址被修改，delete会出错

```
int *p = new int [10];
for(int i = 0; i < 10; i ++) {
      *(p+i) = i;
}
delete []p;
cout<<*(p+1)<<endl;
```

动态空间已经被释放，再访问会出错

动态分配内存

为结构体分配动态内存

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

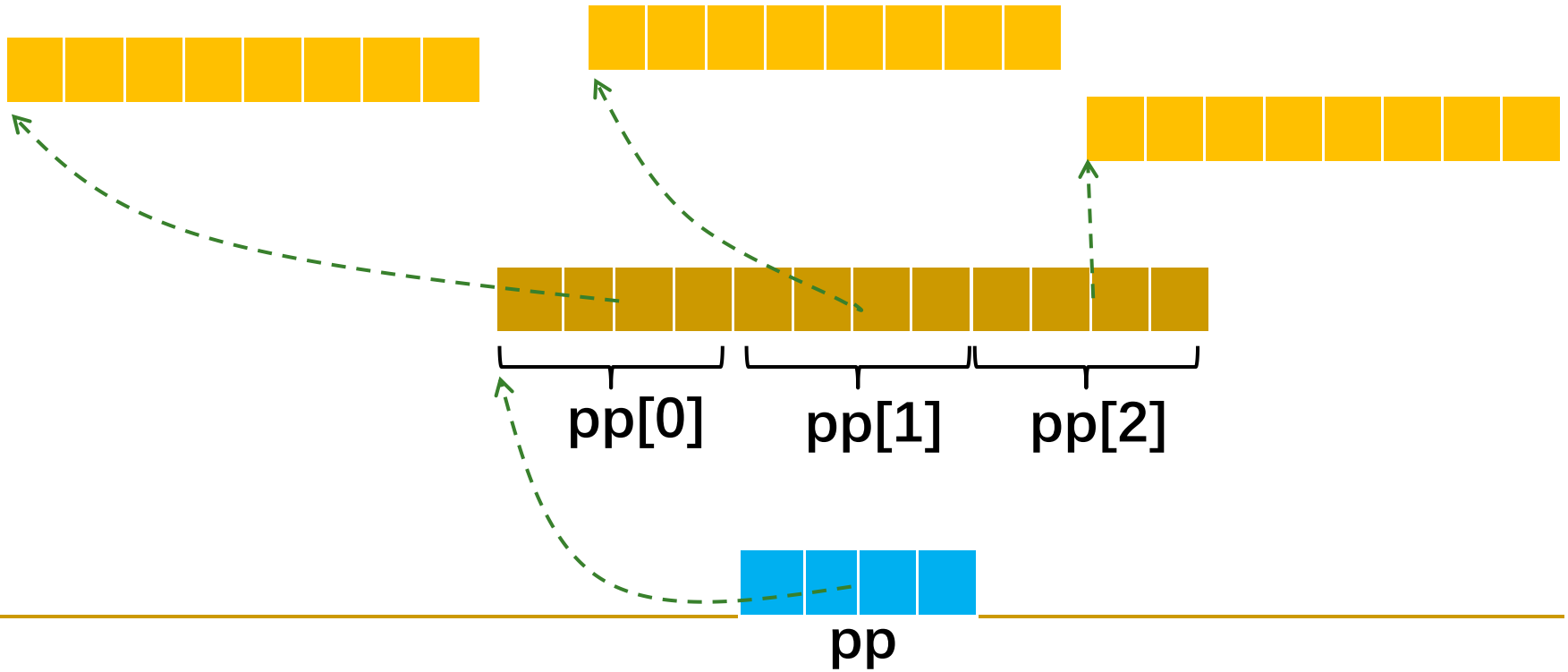
```
student *p = new student[2];
cout<<p[0].id<<p[0].name;
cout<<p->id<<p->name;
```

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

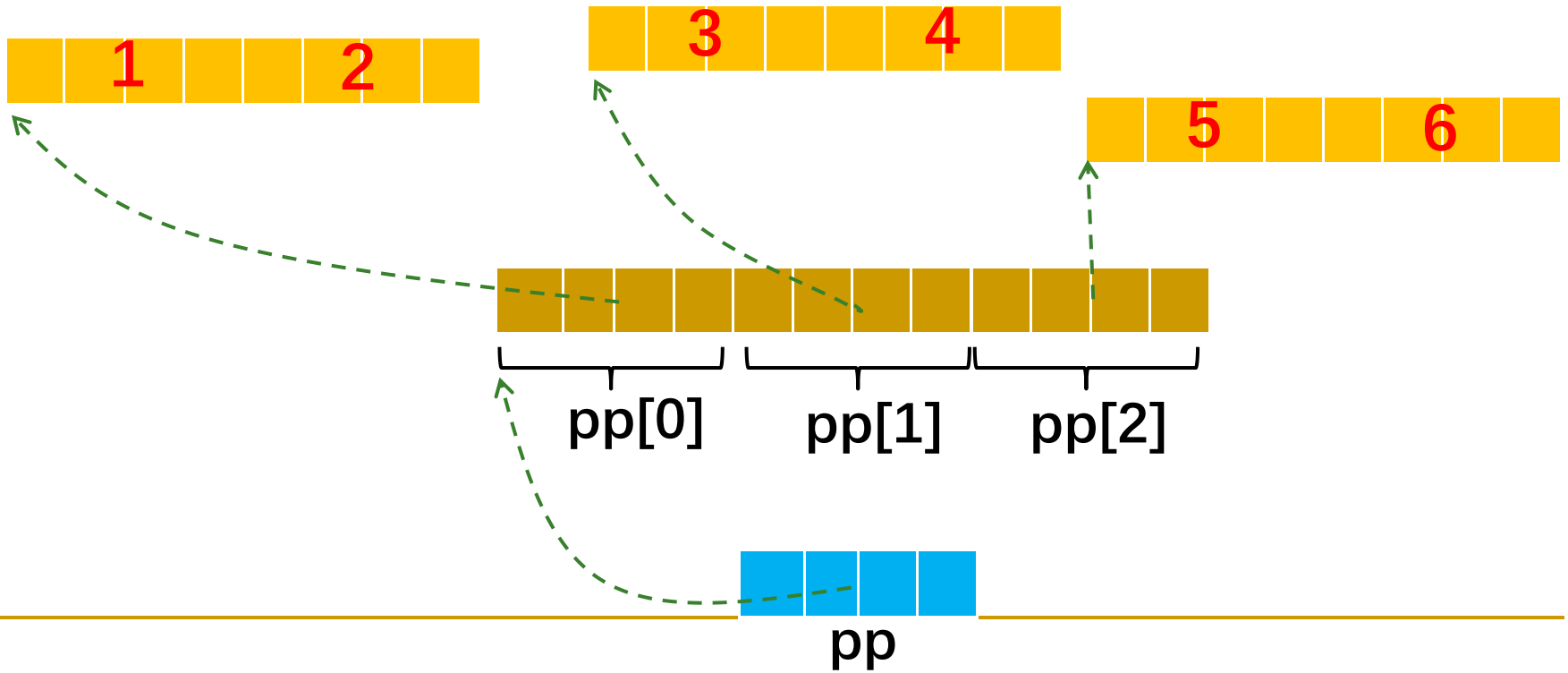
```
int **pp = new int* [m]; //m为变量
for(int i = 0; i < m; i ++) {
        pp[i] = new int [n]; //n为变量
}
```



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

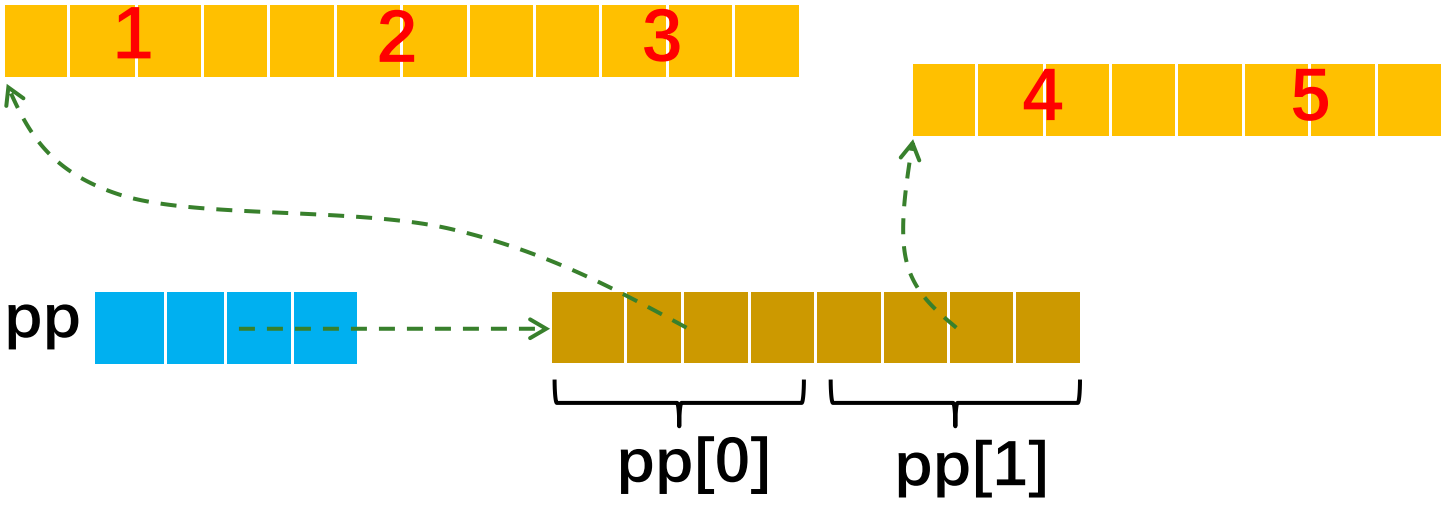
```
int m;
cin>>m;
constant int n = 2;//n必须为常量,不能由用户输入
int (*pp)[n] = new int [m][n];//m可以为变量
```

回收

```
delete []pp;
```

动态生成锯齿数组

```
int **pp = new int* [2];
pp[0] = new int [3];
pp[1] = new int [2];
```



动态生成的数组每一行元素个数不一定相同！

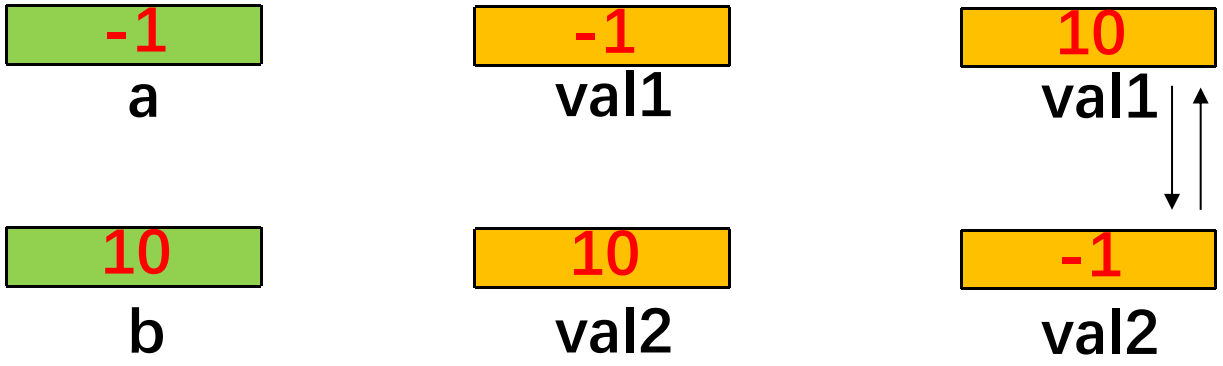
指针与函数

变量作为函数形参

```
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;
}
```

a　　b



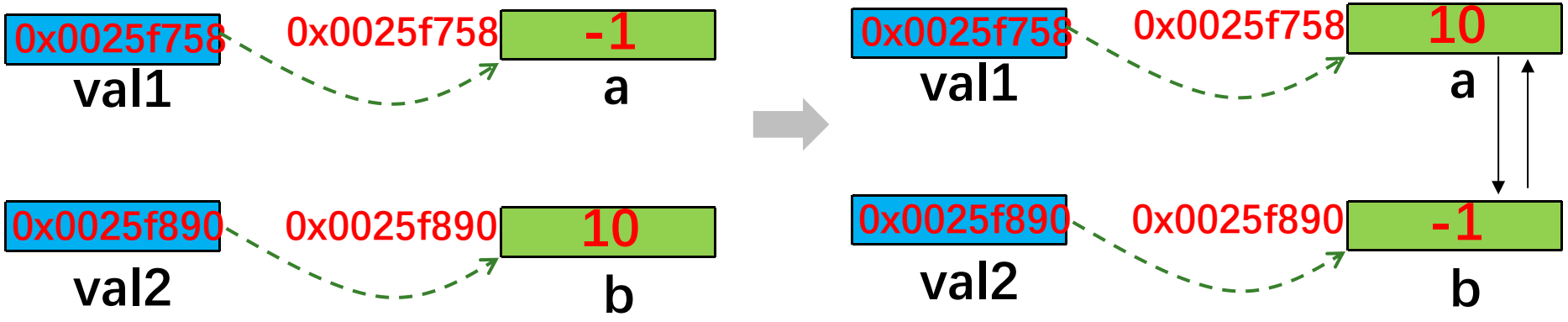
指针与函数

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;
}
```



指针与函数

指针作为函数形参：数组名作为实参

a

```
int main() {
        int a[] = {-1,10,2};

        bubble(a);

        return 0;
}
```

```
void bubble (int *data) {
    for(int i = 3; i > 0; i --) {
        for(int j = 0; j < i; j ++) {
            if(data[j] > data[j+1]) {
                int t = data[j];
                data[j] = data[j+1];
                data[j+1] = t;
            }
        }
    }
}
```

数组名作为实参，传递的是数组首元素的指针

指针与函数

指针作为函数形参：数组名作为实参

a

```
int main() {
        int a[] = {-1,10,2};

        bubble(a, 3);

        return 0;
}
```

```
void bubble (int *data, int n) {
    for(int i = n; i > 0; i --) {
        for(int j = 0; j < i; j ++) {
            if(data[j] > data[j+1]) {
                int t = data[j];
                data[j] = data[j+1];
                data[j+1] = t;
            }
        }
    }
}
```

数组名作为实参，数组大小要通过参数传递！

指针与函数

指针作为函数形参：数组名作为实参

a

```
int main() {
        int a[3] = {-1,10,2};

        bubble(a, 3);

        return 0;
}
```

```
void bubble (int data[], int n) {
    for(int i = n; i > 0; i --) {
        for(int j = 0; j < i; j ++) {
            if(data[j] > data[j+1]) {
                int t = data[j];
                data[j] = data[j+1];
                data[j+1] = t;
            }
        }
    }
}
```

sizeof(data)=?

形参写成数组形式，仍然按照指针处理！

指针与函数

指针作为函数形参：二维数组名作为实参

```
int main() {
        int a[][3] = {{-1,10,2},{0,0,0}};
        func(a, 2, 3);
        return 0;
}
```

```
void func(int b[][3], int m, int n); //ok?
```

```
void func(int (*b)[3], int m, int n);//ok?
```

```
void func(int *b[], int m, int n); //ok?
```

```
void func(int **b, int m, int n); //ok?
```

指针和常量

指针和常量

```
int *p; //指针
```

```
int a = 10;  //a 是变量，可修改
```

```
const int b = 5;  //b 是常量，不能被修改
```

```
p = &a; //ok
```

```
*p = 5; //ok
```

```
p = &b; //error，防止通过指针修改b的值, e.g., *p = 10
```

指针可通过地址修改变量的行为有时候很危险，需加以约束！

常量指针和指针常量

常量指针：不可以通过此指针修改其指向的内容

const <数据类型> * <指针名>

```
const int *p; //常量指针，注意p本身的值可以修改！
int a = 10;  //a 是变量，可修改
const int b = 5;  //b 是常量，不能被修改
p = &b; //ok
*p = 10; //error,不能通过p修改其指向的内容
p = &a; //ok
*p = 5; //error,不能通过p修改其指向的内容
```

常量指针和指针常量

指针常量：指针本身的值不能修改

<数据类型> * const <指针名>

```
int a = 10;  //a 是变量，可修改

const int b = 5;  //b 是常量，不可修改

int * const p = &a; //p必须进行初始化

*p = 10; //ok, a的值可修改

p = &b; //error，p的值不能修改

const int * const p2 = &b; //?
```

指针函数

指针函数：函数的返回值为指针类型

```
int main() {
    int *p;
    p = func(8); //返回大小为8个int的动态空间
    delete []p;
}
int *func(int n) {
    int *p2 = new int[n];
    return p2;
}
```

new所分配的动态空间的生存期是整个程序！

指针函数

指针函数：函数的返回值为指针类型

```
int main() {
    int *p;
    p = func(8); //返回大小为8个int的动态空间
    delete []p;
}
int *func(const int n) {
    int p2[n]; //局部变量
    return p2;
}
```

返回局部变量的地址非常危险！

函数指针

函数指针：指向函数的指针

函数也占据内存空间，也有地址

0x0025f758

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

0x0025f7ff

```
int diff(int a, int b)
{
        return a - b;
}
```

函数名是指向函数首地址的指针常量！

```
cout<<sum<<diff<<endl; //函数首地址
```

函数调用:通过函数名(函数地址)找到函数代码，执行

函数指针

函数指针：指向函数的指针

再看函数声明：

```
int sum (int, int); //函数声明
```

<函数返回类型> <函数名> (形参列表)

函数指针：

<函数返回类型> (*变量名)(形参列表)

```
int (*p) (int, int);
```

指针函数和函数指针

函数指针：指向函数的指针

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

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

```
int (*p)(int, int);
```

```
p = sum;
p(1, 2); //调用sum
(*p)(1,2); //调用sum
```

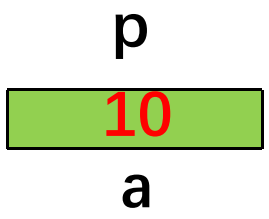
```
p = diff;
p(1, 2); //调用diff
(*p)(1,2); //调用diff
```

引用

引用：变量的别名

引用数据类型 & 引用名 = 变量名

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



变量有两个名字 a, p，
但只有一个空间！

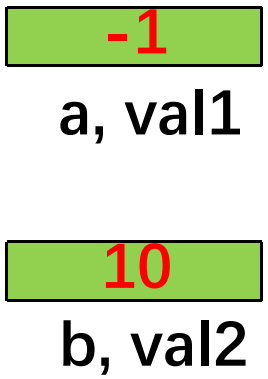
- 引用在声明的同时必须初始化
- 一个引用所引用的对象在初始化后不能改变
- 操作引用和操作所引用的对象一样

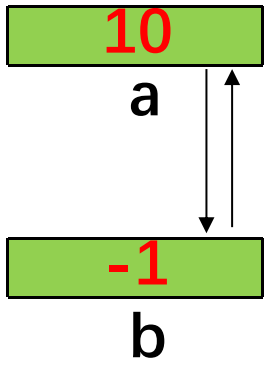
引用

```
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;
}
```

a b





引用

函数返回过程

```
int a = 0;
int main() {
        int b = inc();
        inc() = 1; //ok?
        return 0;
}
```

```
int inc () {
        a++;
        return a;
}
```

全局变量

a

return 时发生什么

```
int inc () {
        a++;
        return a;
}
```

生成一个临时局部变量

a的值copy给临时局部变量

将临时局部变量的值给b，销毁临时局部变量

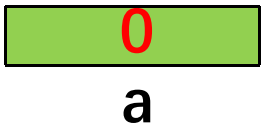
引用

函数返回类型为引用

```
int a = 0, b;
int main() {
        b = inc(a);
        inc(a) = 1; //ok?
        return 0;
}
```

```
int &inc () {
        a++;
        return a;
}
```

全局变量



return 时发生什么

```
int &inc () {
        a++;
        return a;
}
```

返回的就是a本身，可以作为左值！

引用与指针的区别

指针表示的是一个对象变量的地址，而引用则表示一个对象变量的别名。因此在程序中表示其对象变量时，前者要通过取内容运算符“*”，而后者可直接代表.

【例如】

```
int a;      int *pa=&a;      int &ra=a;
```

当要对a赋值123时，下述三个语句是等价的：

```
a=123;      *pa=123;          ra=123;
```

引用与指针的区别

指针是可变的，它可以指向变量a，也可以指向变量b，而引用则只能在建立时一次确定（固定绑定在某一个变量上），不可改变。

```
int a,b=456; int *p=&a; int &ra=a;
p=&b; //将变量b的地址赋给指针p
ra=b; /*将b的值（即456）赋给了与ra绑定的变量a以及引用ra本身，不是将变量b与引用变量ra绑定*/
int &ra=b; /*不合法，为引用ra重新建立新的绑定关系则会导致出现一个编译错误（ra重定义，重复初始化）*/
```

引用与指针的区别

- 由于引用本身不是一个独立的变量（它本身不具有独立的变量地址，使用的是与其绑定的那个变量的地址），所以，不能出现引用的引用，不能出现元素为引用的数组，也不能使用指向引用的指针
- 指针是独立变量，可以出现指针的指针、可以出现元素为指针的数组，也可以说明对指针的引用

举例

```
int &&ref;  //ERR! 不能出现引用的引用
int &refa[10]; //ERR! 不能出现元素为引用的数组
int &*refp;//ERR! 不能使用指向引用的指针
int *pi, *&pref=pi; /*OK! 可以说明对指针的引用，将引用pref与int*类型的指针变量pi进行了绑定*/
```

举例

```
#include <iostream.h>
void main(){
  int a;
  int &ra=a; //ra为引用，它是变量a的“别名”
  int *pa=&a;
  cout<<"&a ="<<&a<<endl;
  //a为独立变量，具有独立地址
  cout<<"&ra="<<&ra<<endl;
  //ra是a的“别名”，不具有独立地址
  cout<<"&pa="<<&pa<<endl;
  //pa为独立变量，具有独立地址
  a=123;
  //对a赋值123后，使ra及*pa的值都成为123
```

举例

```
    cout<<"a="<<a<<endl;
    cout<<"*pa="<<*pa<<endl;
    cout<<"ra="<<ra<<endl;
    *pa=456; //对*pa赋值456后，使a及ra的值都成为456
    cout<<"a="<<a<<endl;
    cout<<"*pa="<<*pa<<endl;
    cout<<"ra="<<ra<<endl;
    ra=789; //对ra赋值789后，使a及*pa的值都成为789
    cout<<"a="<<a<<endl;
    cout<<"*pa="<<*pa<<endl;
    cout<<"ra="<<ra<<endl;
}
```

举例

程序运行结果：

```
&a =0x0065FDF4
&ra=0x0065FDF4
&pa=0x0065FDF0
a=123
*pa=123
ra=123
a=456
*pa=456
ra=456
a=789
*pa=789
ra=789
```

END