第六章 指针

C++简明双链教程（李昕著，清华大学出版社）

作者：李昕

PPT制作者：刘雯

目录

01 指针的概念与指针变量的定义

1.1 指针与指针变量

1. 指针的作用

指针是C/C++的灵魂，正确灵活地运用它，可以有效地表达一些复杂的数据结构，如系统地动态分配内存、消息机制、任务调度、灵活矩阵定时等，掌握指针可以使程序更加简洁、紧凑、高效。

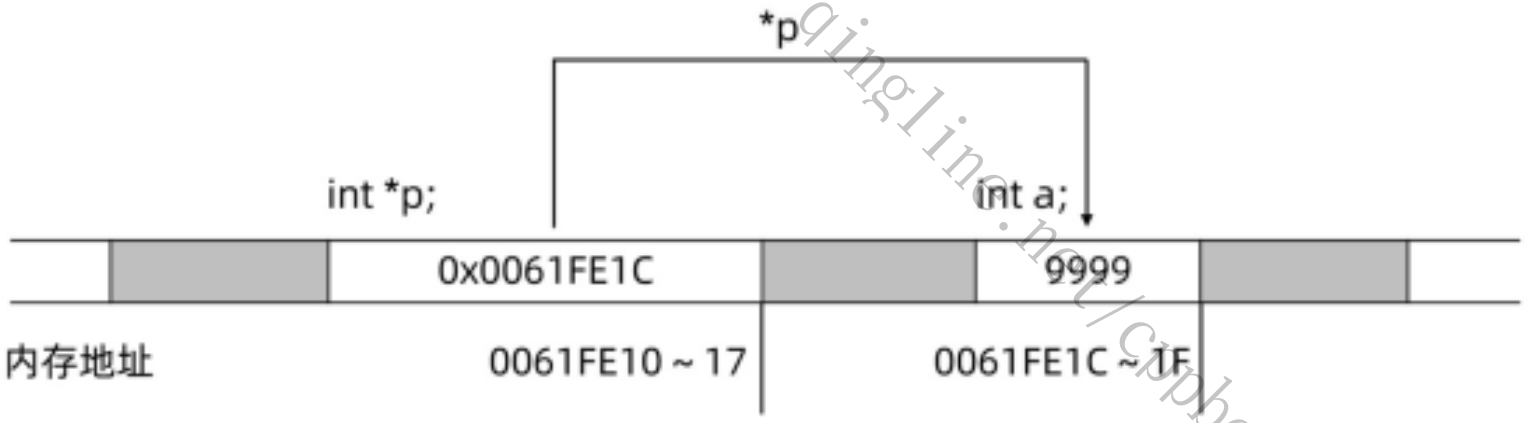
2. 如何理解指针

关键是要理解地址的概念，而指针只是一个存储地址的变量。也就是说，指针也是一个变量，其中存储的“值”是一个地址，而这个地址是某个普通变量的地址，通过地址可以间接的找到需要处理的普通变量。

1.1 指针与指针变量

指针变量：专门存放变量地址的变量。

如图，变量 a 的地址是 0x61FE1C ， 指针变量 p 存放的是变量 a 的地址。此时说 p 指向了 a 。



知识点

索引	要点	正链	反链
T611	掌握指针基本概念，它是一个存储地址的变量，关键是要理解地址的概念	T341	

1.2 指针变量的定义

数据类型 *指针名；

如：int *p,*q,a,b,c; char *s,ch,s; float *y,*w,x;

其中：p,q,s,y,w为指针变量，a,b,c,ch,s,x为普通变量。

数据类型是指针变量的目标类型，即它所指变量的类型。此处的*是告诉计算机，其后的变量为**指针变量**，不包含任何的运算。

1.2 指针变量的定义

int *p,*q,a,b,c;　char *s,ch,s;　float *y,*w,x;

空格应该加在*的左侧还是右侧?

空格在程序中所起到的最大作用是**分隔**，当书写int a;时，空格将数据类型和变量名进行了分隔。而在指针定义时，*可以起到同样的分隔作用，因此**可以在左侧写空格，也可以在右侧写空格，也可以左右都不写空格**，都是正确的。

1.2 指针变量的定义

两个有关的运算符：

&：取地址运算符，只能作用于变量。

*：取内容运算符（或称“间接访问”运算符）。

例如：&a为取变量a的地址，*p为取指针p所指向的存贮单元的内容。

1.2 指针变量的定义

```
1   #include <iostream>
2   using namespace std;
3   int main ()
4   {
5       int a=97;
6       int *p;
7       p=&a;
8       cout<<(&a)<<'\t'<<p<<endl;
9       cout<<a<<'\t'<<*p<<endl;
10      *p=65;
11      cout<<a<<'\t'<<*p<<endl;
12      return 0;
13  }
```

样例输出

```
0x61fe14    0x61fe14
97      97
65      65
```

1. 第6行定义了一个指针，为了与变量a进行绑定，其类型必须与a保持一致，因此定义为整型指针。

2. 第7行将变量a的地址赋值给p，指针变量只能用地址进行赋值，而且跟其他变量一样，指针必须先赋值再使用。

3. 第8行的输出结果显示，p中存放了变量a的地址。

1.2 指针变量的定义

```
1   #include <iostream>
2   using namespace std;
3   int main ()
4   {
5       int a=97;
6       int *p;
7       p=&a;
8       cout<<(&a)<<'\t'<<p<<endl;
9       cout<<a<<'\t'<<*p<<endl;
10      *p=65;
11      cout<<a<<'\t'<<*p<<endl;
12      return 0;
13  }
```

样例输出

```
0x61fe14    0x61fe14
97      97
65      65
```

4. 第9行使用了取内容运算符*，可以看到a和*p的输出内容相同。这是因为p指向了a，即p中存放了变量a的地址，*p表示通过地址定位了变量a，然后将a的值进行了打印。

5. 第10-11行显示当修改*p后，a的值同时进行了改变。因为*p和a完全等价，它们操作的是同一块内存区，即变量a的内存区。

6. 再次强调相同运算符在定义语句和非定义语句中含义是完全不同的。第6行中的*表示p是一个指针变量，第9-11行中的*表示取指针所指向区域的内容，即取a的值。

知识点

索引	要点	正链	反链
T612	掌握指针变量的定义方式，区分取地址运算符&和取内容运算符*。理解*在定义语句和非定义语句含义的不同。	T277	
T613	指针与所指向变量共享存储空间，如果把指针作为函数参数，则形参与实参共享存储空间。因此很多时候把指针作为形参是为了获取函数中的计算结果，也就是达到了多返回值的目的	T336	

1.3 指针的两个“值”

指针作为一个变量，它存储了一个“值”，其实就是一个**地址**；

但通过操作符*，它可以得到**它指向变量的值**，称为间接取值。

地址相当于一个ID，帮助指针找到对应的变量。对于这个地址，不需要关心它具体的值是多少，**重点关注它所指向的内容。**

例如，当去图书馆找一本书时，每本书都有编号，代表哪个房间几号柜第几层。通过这个编号，可以快速找到书，书是我们关注的实体，但编号的具体内容对于我们没有使用价值，只是一个帮助定位的媒介。

1.3 指针的两个“值”

```
1   #include <iostream>
2   using namespace std;
3   int main ()
4   {
5       int a=10, b=20, *p1=&a, *p2=&b;
6       cout<<a<<'\t'<<*p1<<endl<<b<<'\t'<<*p2<<endl<<endl;
7       int *p = p1;
8       p1 = p2;
9       p2 = p;
10      cout<<a<<'\t'<<*p1<<endl<<b<<'\t'<<*p2<<endl<<endl;
11      a=10, b=20, p1=&a, p2=&b;
12      int temp = *p1;
13      *p1 = *p2;
14      *p2 = temp;
15      cout<<a<<'\t'<<*p1<<endl<<b<<'\t'<<*p2<<endl<<endl;
16      return 0;
17  }
```

样例输出

```
10      10
20      20

10      20
20      10

20      20
10      10
```

1. 第5行将p1指向了a，p2指向了b，因此*p1和a等价，*p2和b等价。

2.第7-9行将两个指针所存储的地址发生了交换，即让p1指向了b，p2指向了a，因此*p1和*p2显示的内容发生了交换，但是a和b并没有发生任何改变。

1.3 指针的两个“值”

```
1   #include <iostream>
2   using namespace std;
3   int main ()
4   {
5       int a=10, b=20, *p1=&a, *p2=&b;
6       cout<<a<<'\t'<<*p1<<endl<<b<<'\t'<<*p2<<endl<<endl;
7       int *p = p1;
8       p1 = p2;
9       p2 = p;
10      cout<<a<<'\t'<<*p1<<endl<<b<<'\t'<<*p2<<endl<<endl;
11      a=10, b=20, p1=&a, p2=&b;
12      int temp = *p1;
13      *p1 = *p2;
14      *p2 = temp;
15      cout<<a<<'\t'<<*p1<<endl<<b<<'\t'<<*p2<<endl<<endl;
16      return 0;
17  }
```

样例输出

```
10      10
20      20

10      20
20      10

20      20
10      10
```

3.第12-14行将两个指针所指向的内容进行了交换，即交换了a和b，但p1仍旧指向了a，p2仍旧指向了b。

知识点

索引	要点	正链	反链
T614	注意区分指针包含的两个“值”：存储的地址以及访问地址所指向空间的值		

1.4 强大的指针

1. 既然可以通过变量访问简单直观的访问内存，为什么需要用指针间接访问内存呢?

因为在很多情况下，是**找不到变量**的，但是所有程序运行后都会加载到内存上，可以通过一些方法定位到所需要处理的内容在内存上的地址。

2. 举例:

打游戏的时候可以通过对比血量发生变化前后的两块内存区域，找到变化的地方，也就找到了代表血量的内存区，这时候你就可以自己写一个程序，用指针修改找到的内存区域的值，让你的英雄“满血复活”。

很多系统将存储真实密码的变量和存储你输入密码的变量是连续定义的，这样它们的内存区是相邻的。通过改变输入，很快就可以定位输入密码的内存位置，将其相邻内存区域的内容复制过来，就可以通过密码验证了。

1.4 强大的指针

3. 指针是一把双刃剑

从以上案例可以看出，**通过指针可以直接访问内存区**，造成了指针的强大功能，这是Java，Python等没有指针的编程语言所无法办到的。但是同样指针也是一把双刃剑，对不确定内存区域进行修改，会造成程序的错误，甚至系统的崩溃。因此使用指针一定要特别小心，**保证指针一定要指向正确的内存区**。只要对内存分布、地址编码有了充分的了解，就可以发挥指针的强大功能。

1.4 强大的指针

4. **对于在线评测的算法试题而言，即使完全不懂指针，也不影响题目求解。**

可以用引用代替指针；

可以用string类型代替字符数组来避免使用指针访问字符数组。

知识点

索引	要点	正链	反链
T615	理解指针的优势和弊端，使用指针的时候要保证指针指向正确的内存区		

02 数组与指针

2.1 一维数组与指针

数组名不是指针，但大多数使用到数组名的地方，编译器都会把数组名隐式转换成一个指向数组首元素的指针常量来处理。也就是说，**数组名被解释为存放地址的变量**，但却是常变量，**其中保存的地址不可以被修改**。

除了存储的地址不可以被修改外，数组名就是一个指针。数组通过下标访问元素，跟指针进行偏移进行元素访问是完全相同的。这种对于连续空间的数据，通过偏移访问任意元素的方式，也称为随机访问。

2.1 一维数组与指针

以int a[5],*p=a; 为例，下表中前4列完全等价，后3列完全等价。

元素的值				元素的地址		
a[0]	*a	*p	p[0]	a	p	&a[0]
a[1]	*(a+1)	*(p+1)	p[1]	a+1	p+1	&a[1]
a[2]	*(a+2)	*(p+2)	p[2]	a+2	p+2	&a[2]
a[3]	*(a+3)	*(p+3)	p[3]	a+3	p+3	&a[3]
a[4]	*(a+4)	*(p+4)	p[4]	a+4	p+4	&a[4]

2.1 一维数组与指针

有两种特例情况数组名**不会被转换为指针**，以 int a[10]; 为例：

1）对数组名使用**sizeof**运算符，则sizeof(a)代表整个数组所占的内存大小，a是长度为10的int（4字节）数组，运算结果是40。用数组的总字节长度除以单个元素的字节长度得到元素个数，例如sizeof(a)/sizeof(*a)。

2）对**数组名取地址**，&a表示数组的地址。注意，数组的地址和数组首元素的地址是不同的概念，尽管二者存储的值是相同的，但它们的跨度是不同的。这就像山东省的省政府在济南，但是济南的市政府也在济南，地址相同，但是代表的意义完全不同。

2.1 一维数组与指针

```
1   int main (  )
2   {
3       int a[10];
4       cout<<a<<'\t'<<&a<<endl;
5       cout<<a+1<<'\t'<<&a+1<<endl;
6       return 0;
7   }
```

样例输出

```
0x61fdf0 0x61fdf0
0x61fdf4 0x61fe18
```

a 和 &a的存储的地址相同。

a 指向首元素，右移一位，地址增加了4字节，也就是一个int的长度；

&a 指向数组，右移一位，地址增加了40字节，相当于指向了下一个数组（可能并不存在），这在C++里称为尾后指针。

2.1 一维数组与指针

```
1   int main (  )
2   {
3       int a[10];
4       cout<<a<<'\t'<<&a<<endl;
5       cout<<a+1<<'\t'<<&a+1<<endl;
6       return 0;
7   }
```

样例输出

```
0x61fdf0 0x61fdf0
0x61fdf4 0x61fe18
```

除了前面说的两种例外，其他情况下编译器都将**数组名隐式转换成指针常量**。

如： a[3] ⟶ *(a + 3)

因为第一种写法会自动转换成第二种，这个过程需要一些开销，所以第二种写法通常效率会高一些。但是毕竟第一种写法更方便，通常代码里不会有太多的下标引用数组，因此这种效率提高可以忽略。

数组和指针在C/C++中可以认为是相同内容的两种不同书写形式，除了数组名是常指针不可被修改外，二者是完全一样的。数组可以写成指针形式，同样指针也可以用数组方式进行访问。

2.1 一维数组与指针

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

int main (  )
{
	int a[5]={1,2,3,4,5};
	int*p = a+2;
	p[1] = 7;
	for(auto e:a)
		cout<<e<<' ';
	return 0;
}
```

样例输出
1 2 3 7 5

指针p指向a[2]的地址；

p[1]=*(p+1)=*((a+2)+1)=a[3];

因此a[3]对应的值被修改。

2.1 一维数组与指针

指针可以进行**类型转换**，然后按照新的数据类型访问空间。

以**memset**为例，它可以对大块内存进行赋值，但是只能以字节为单位，因此无论任意类型的数组，在使用memset时都是以字节为单位进行进行赋值的。

memset函数需要**头文件**<memory.h>或<string.h>；

函数原型是extern void *memset(void *buffer, int c, int count)；

其中buffer为指针或是数组，c是赋给buffer的值，count是buffer的长度。

2.1 一维数组与指针

```
1   #include<bits/stdc++.h>
2   using namespace std;
3   int main() {
4       int arr[5];
5       memset(arr,0,sizeof(arr));
6       for(auto i:arr)     cout<<i<<" "; cout<<endl;
7       memset(arr,255,sizeof(arr));
8       for(auto i:arr)     cout<<i<<" "; cout<<endl;
9       memset(arr,1,sizeof(arr));
10      for(auto i:arr)     cout<<i<<" "; cout<<endl;
11      cout<<bitset<32>(16843009)<<endl;
12      char* s=(char*)arr;
13      for(int i=0;i<sizeof(arr)/sizeof(char);i++)     s[i]=1;
14      for(auto i:arr)     cout<<i<<" "; cout<<endl;
15  }
```

样例输出

```
0 0 0 0 0
-1 -1 -1 -1 -1
16843009 16843009 16843009 16843009 16843009
00000001000000010000000100000001
16843009 16843009 16843009 16843009 16843009
```

- 第5行将整个数组全部赋值为0
- 第7行将数组的所有内存区间设置为全1

2.1 一维数组与指针

```
1   #include<bits/stdc++.h>
2   using namespace std;
3   int main() {
4       int arr[5];
5       memset(arr,0,sizeof(arr));
6       for(auto i:arr)     cout<<i<<" "; cout<<endl;
7       memset(arr,255,sizeof(arr));
8       for(auto i:arr)     cout<<i<<" "; cout<<endl;
9       memset(arr,1,sizeof(arr));
10      for(auto i:arr)     cout<<i<<" "; cout<<endl;
11      cout<<bitset<32>(16843009)<<endl;
12      char* s=(char*)arr;
13      for(int i=0;i<sizeof(arr)/sizeof(char);i++)     s[i]=1;
14      for(auto i:arr)     cout<<i<<" "; cout<<endl;
15  }
```

样例输出

```
0 0 0 0 0
-1 -1 -1 -1 -1
16843009 16843009 16843009 16843009 16843009
00000001000000010000000100000001
16843009 16843009 16843009 16843009 16843009
```

- 3. 但第10行输出结果显示，并没有按照预期将5个元素设置为1，因为memset以字节为单位，每个字节都被赋值为1。
- 4. 第12-14行展示了与第9行相同执行的效果。

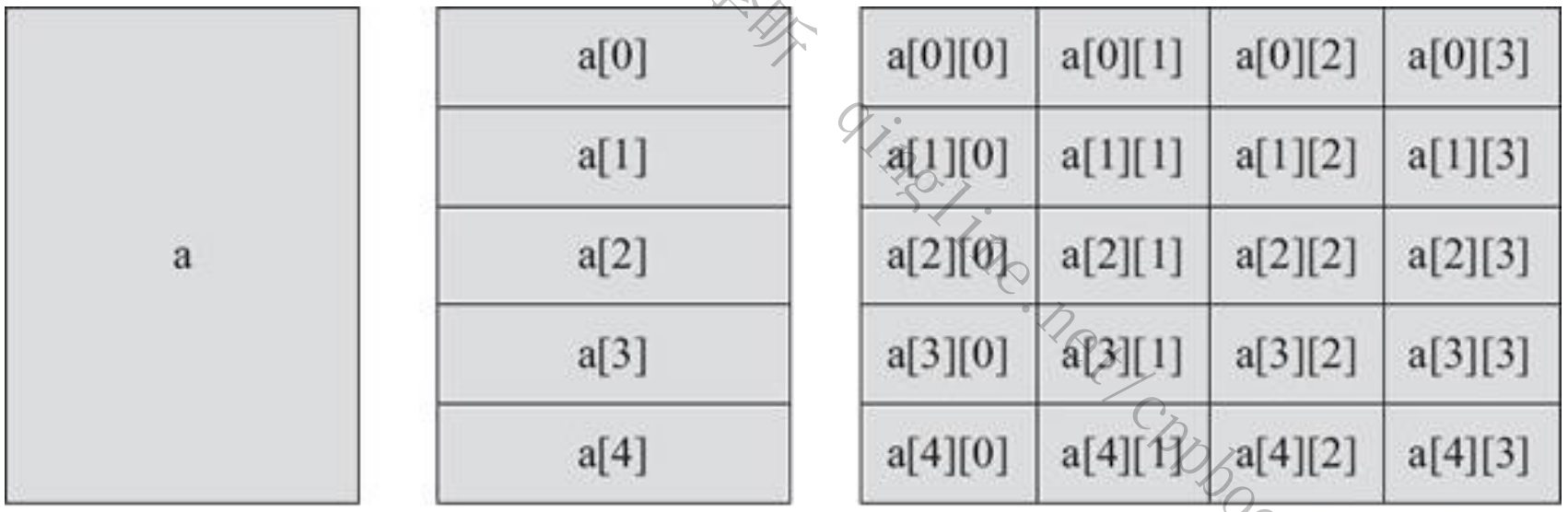
知识点

索引	要点	正链	反链
T621	理解数组名和指针的异同，理解表6.1中各种方式的对应关系，对于一个数组空间，即使给出的是指针形式，也可以按照数组形式进行访问，更便于理解	T513,T823	T626
T622	通过显示转换指针的类型，就可以对空间采用不同的访问形式	T252	

2.2 二维数组与指针

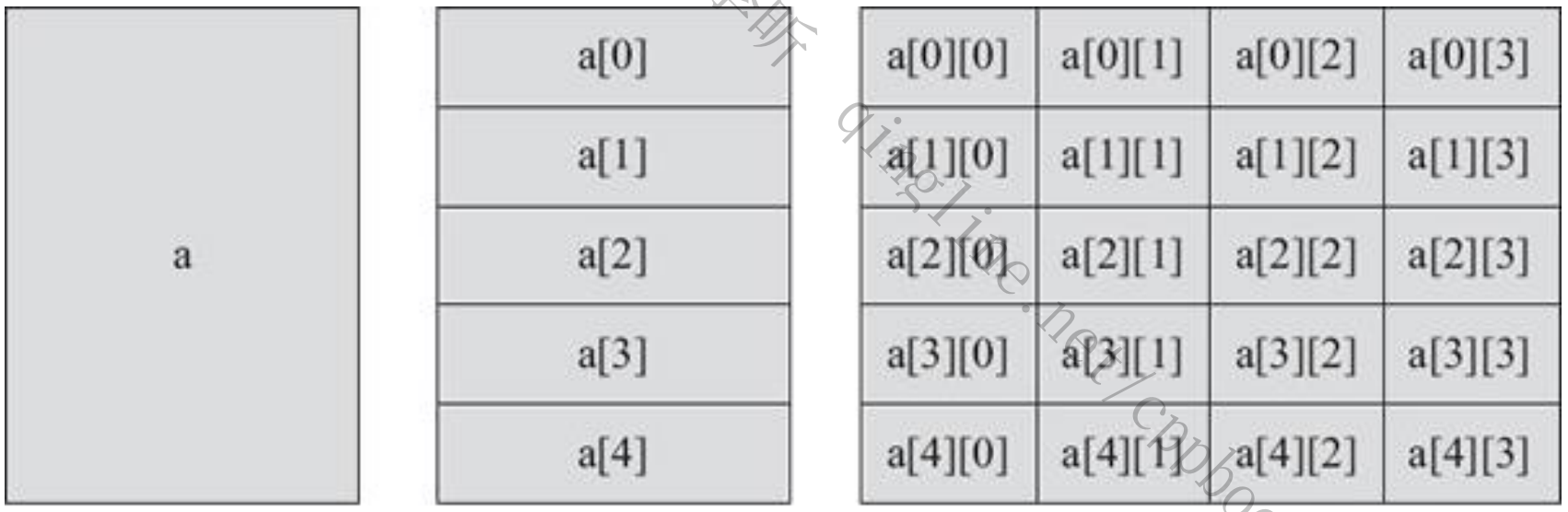
二维数组及多维数组的存储原理与一维数组一样。

以一个 5 行 4 列的二维数组**int a[5][4];**为例，它在逻辑上是由行和列组成的，可以分为**三层**来理解，如图所示。



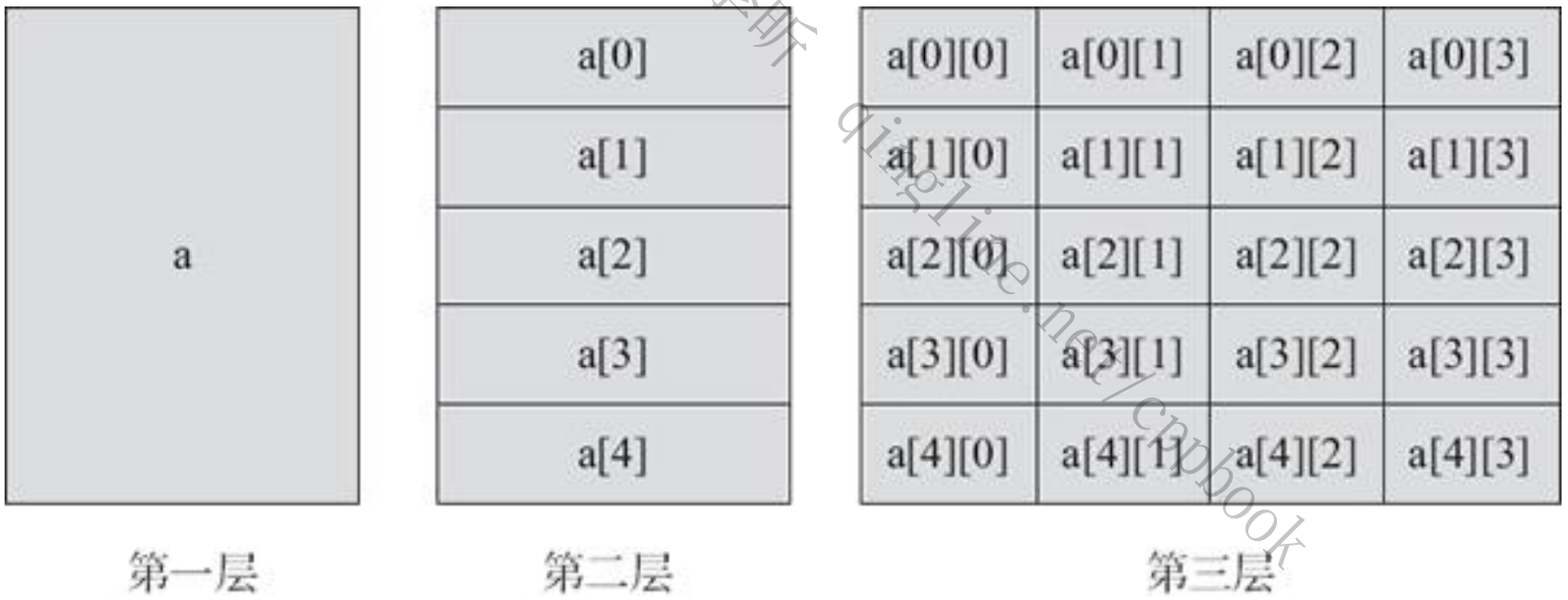
2.2 二维数组与指针

第一层，将数组 a 看作一个**变量**，该变量的地址为 &a，长度为 sizeof(a)。因为数组的长度为元素数量乘以每个元素类型的大小，这里的二维数组 a 为 5 行 4 列共 20 个元素，每个元素占用 4 字节，所以变量 a 占用 80 字节。



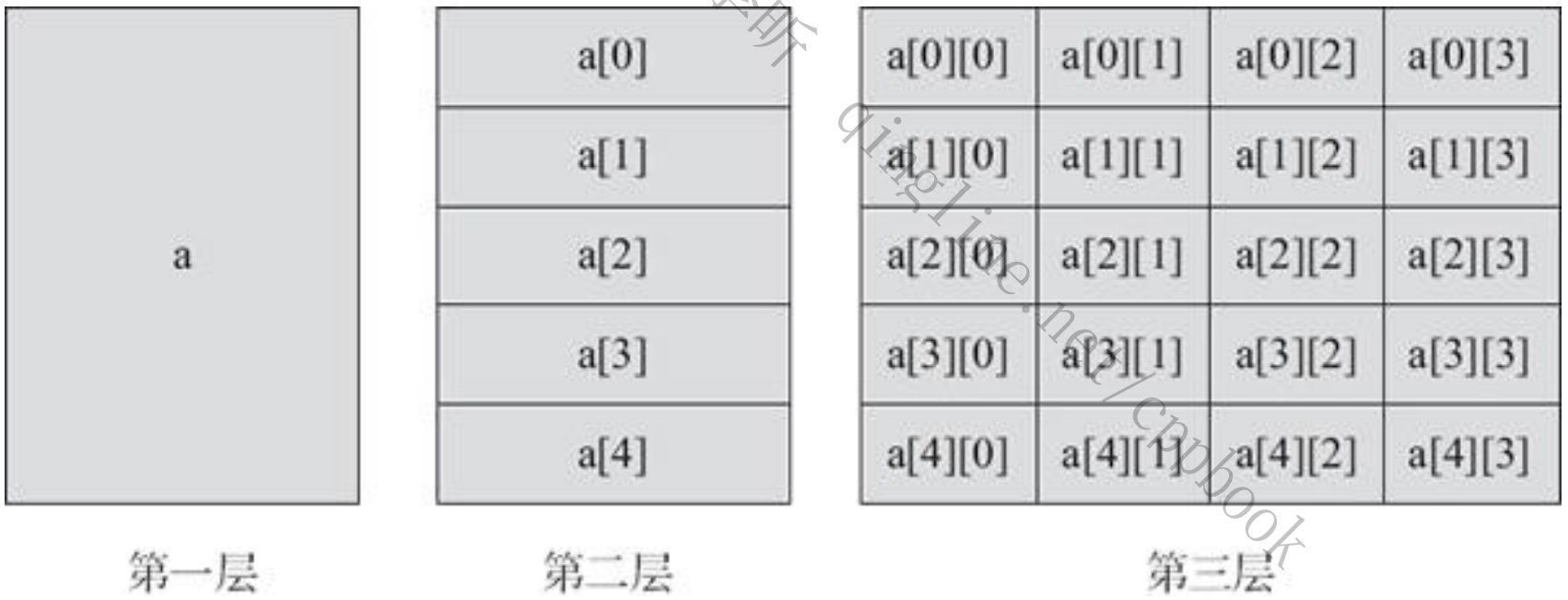
2.2 二维数组与指针

第二层，将数组 a 看作一个**一维数组**，由 a[0]、a[1]、a[2]、a[3] 与 a[4] 等 5 个元素组成。数组的首地址为 a 或 &a[0]（即数组首地址和第一个元素的地址相同，而每个数组元素的地址相差为 16，表示每个数组元素的长度为 16），使用 sizeof(a[0]) 可得到数组元素的长度。



2.2 二维数组与指针

第三层，将第二层中的**每个数组元素看作一个单独的数组**。第二层中的每一个元素又由 4 个元素构成，如 a[0] 又由 a[0][0]、a[0][1]、a[0][2] 与 a[0][3] 等 4 个元素组成，每个元素占用sizeof(a[0][0]) =4字节。



2.2 二维数组与指针

```cpp
#include<iostream>
#include<typeinfo>
using namespace std;
#define type(obj) typeid(obj).name()

int main (  )
{
    int a[5][4];
    cout<<"对象\ta\ta[0]\ta[0][0]"<<endl;
    cout<<"类型"<<'\t'<<type(a)<<'\t'<<type(a[0])<<'\t'<<type(a[0][0])<<endl;
    cout<<"地址类型"<<'\t'<<type(&a)<<'\t'<<type(&a[0])<<'\t'<<type(&a[0][0])<<endl;
    cout<<"地址"<<'\t'<<&a<<'\t'<<&a[0]<<'\t'<<&a[0][0]<<endl;
    cout<<"空间"<<'\t'<<sizeof(a)<<'\t'<<sizeof(a[0])<<'\t'<<sizeof(a[0][0])<<endl;
    cout<<"地址+1"<<'\t'<<&a+1<<'\t'<<&a[0]+1<<'\t'<<&a[0][0]+1<<endl;
    cout<<"地址空间"<<'\t'<<sizeof(&a)<<'\t'<<sizeof(&a[0])<<'\t'<<sizeof(&a[0][0])<<endl;

    return 0;
}
```

样例输出			
对象	a	a[0]	a[0][0]
类型	A5_A4_i	A4_i	i
地址类型	PA5_A4_i	PA4_i	Pi
地址	0x61fdc0	0x61fdc0	0x61fdc0
空间	80	16	4
地址+1	0x61fe10	0x61fdd0	0x61fdc4
地址空间	8	8	8

将三层元素的相关信息进行输出比较，观察它们的异同。

2.2 二维数组与指针

```
1  #include<iostream>
2  #include<typeinfo>
3  using namespace std;
4  #define type(obj) typeid(obj).name()
5
6  int main (  )
7  {
8      int a[5][4];
9      cout<<"对象\ta\ta[0]\ta[0][0]"<<endl;
10     cout<<"类型"<<'\t'<<type(a)<<'\t'<<type(a[0])<<'\t'<<type(a[0][0])<<endl;
11     cout<<"地址类型"<<'\t'<<type(&a)<<'\t'<<type(&a[0])<<'\t'<<type(&a[0][0])<<endl;
12     cout<<"地址"<<'\t'<<&a<<'\t'<<&a[0]<<'\t'<<&a[0][0]<<endl;
13     cout<<"空间"<<'\t'<<sizeof(a)<<'\t'<<sizeof(a[0])<<'\t'<<sizeof(a[0][0])<<endl;
14     cout<<"地址+1"<<'\t'<<&a+1<<'\t'<<&a[0]+1<<'\t'<<&a[0][0]+1<<endl;
15     cout<<"地址空间"<<'\t'<<sizeof(&a)<<'\t'<<sizeof(&a[0])<<'\t'<<sizeof(&a[0][0])<<endl;
16
17     return 0;
18 }
```

样例输出			
对象	a	a[0]	a[0][0]
类型	A5_A4_i	A4_i	i
地址类型	PA5_A4_i	PA4_i	Pi
地址	0x61fdc0	0x61fdc0	0x61fdc0
空间	80	16	4
地址+1	0x61fe10	0x61fdd0	0x61fdc4
地址空间	8	8	8

typeid(obj).name()可以输出变量的类型，第4行将其定义为**宏**，简化后继代码中的书写。宏是一个替换的概念，所有出现type(obj)的内容都在机器代码中被编译器替换为typeid(obj).name()。

2.2 二维数组与指针

- typeid(obj).name()的输出中，**i表示int**，**A表示数组**，**P表示指针。**
- 三层元素的首地址相同，但是它们的类型并不相同。
  - 进一步可以推导出地址也是类型的。
  - 例如：65和'A'在内存中完全相同，但是类型不同。

样例输出			
对象	a	a[0]	a[0][0]
类型	A5_A4_i	A4_i	i
地址类型	PA5_A4_i	PA4_i	Pi
地址	0x61fdc0	0x61fdc0	0x61fdc0
空间	80	16	4
地址+1	0x61fe10	0x61fdd0	0x61fdc4
地址空间	8	8	8

从表格的类型一行可以看出，a是一个5行4列的整型数组，a[0]是一个包含4个元素的一维整型数组，而a[0][0]是一个整数。它们对应的地址是指针类型。

2.2 二维数组与指针

```
1  #include<iostream>
2  #include<typeinfo>
3  using namespace std;
4  #define type(obj) typeid(obj).name()
5
6  int main (  )
7  {
8      int a[5][4];
9      cout<<"对象\ta\ta[0]\ta[0][0]"<<endl;
10     cout<<"类型"<<'\t'<<type(a)<<'\t'<<type(a[0])<<'\t'<<type(a[0][0])<<endl;
11     cout<<"地址类型"<<'\t'<<type(&a)<<'\t'<<type(&a[0])<<'\t'<<type(&a[0][0])<<endl;
12     cout<<"地址"<<'\t'<<&a<<'\t'<<&a[0]<<'\t'<<&a[0][0]<<endl;
13     cout<<"空间"<<'\t'<<sizeof(a)<<'\t'<<sizeof(a[0])<<'\t'<<sizeof(a[0][0])<<endl;
14     cout<<"地址+1"<<'\t'<<&a+1<<'\t'<<&a[0]+1<<'\t'<<&a[0][0]+1<<endl;
15     cout<<"地址空间"<<'\t'<<sizeof(&a)<<'\t'<<sizeof(&a[0])<<'\t'<<sizeof(&a[0][0])<<endl;
16
17     return 0;
18 }
```

样例输出			
对象	a	a[0]	a[0][0]
类型	A5_A4_i	A4_i	i
地址类型	PA5_A4_i	PA4_i	Pi
地址	0x61fdc0	0x61fdc0	0x61fdc0
空间	80	16	4
地址+1	0x61fe10	0x61fdd0	0x61fdc4
地址空间	8	8	8

每个类型占据的空间不同，a占据5*4*sizeof(int)=80个字节，而a[0]占据4*sizeof(int)=20个字节，a[0][0]只占据4个字节，因此地址进行+1操作时，得到的结果完全不同。

2.2 二维数组与指针

```
#include<iostream>
#include<typeinfo>
using namespace std;
#define type(obj) typeid(obj).name()

int main (  )
{
    int a[5][4];
    cout<<"对象\ta\ta[0]\ta[0][0]"<<endl;
    cout<<"类型"<<'\t'<<type(a)<<'\t'<<type(a[0])<<'\t'<<type(a[0][0])<<endl;
    cout<<"地址类型"<<'\t'<<type(&a)<<'\t'<<type(&a[0])<<'\t'<<type(&a[0][0])<<endl;
    cout<<"地址"<<'\t'<<&a<<'\t'<<&a[0]<<'\t'<<&a[0][0]<<endl;
    cout<<"空间"<<'\t'<<sizeof(a)<<'\t'<<sizeof(a[0])<<'\t'<<sizeof(a[0][0])<<endl;
    cout<<"地址+1"<<'\t'<<&a+1<<'\t'<<&a[0]+1<<'\t'<<&a[0][0]+1<<endl;
    cout<<"地址空间"<<'\t'<<sizeof(&a)<<'\t'<<sizeof(&a[0])<<'\t'<<sizeof(&a[0][0])<<endl;

    return 0;
}
```

样例输出			
对象	a	a[0]	a[0][0]
类型	A5_A4_i	A4_i	i
地址类型	PA5_A4_i	PA4_i	Pi
地址	0x61fdc0	0x61fdc0	0x61fdc0
空间	80	16	4
地址+1	0x61fe10	0x61fdd0	0x61fdc4
地址空间	8	8	8

最后一行非常让初学者迷惑。这是因为地址本身也需要占用空间，在64位系统中，每个地址占64/8=8个字节，因此64位系统最大能管理264的内存空间。

知识点

索引	要点	正链	反链
T623	掌握二维数组的存储原理，理解多级地址形成指针的区别	T531	T624

2.3 二维数组与一维数组

计算机中的内存是按照**线性规划地址**的，也就是说**所有的地址都是一维的**。因此二维以上的高维数组只是逻辑上的概念，在最终存储时，还是一维的。因为数组要求所有元素的存储空间必须是**连续的**，所以高维数组都是低维数据存储结束后，存储下一个低维的所有数据，所有数据在一维上是连续的。

2.3 二维数组与一维数组

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

int main ()
{
    int a[3][4] = { {3,5,7,6}, {1,8,2,5},{7,6,2,9}};
    int *p=&a[0][0];
    for ( int i = 0; i < 3; i++ ){
        for ( int j = 0; j < 4; j++ ){
            cout <<'\t'<< p[i*4+j];
        }
        cout<<endl;
    }
    return 0;
}
```

样例输出

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

以二维数组a[3][4]为例，C/C++是行优先存储的，首先存储第0行的4个元素，然后继续存储第1行的4个元素，最后存储第2行的4个元素，这12个元素在地址空间上是连续的。也就是说，高维数组也可以转换为一维数组进行访问。

2.3 二维数组与一维数组

```
1   #include <iostream>
2   using namespace std;
3
4   int main ()
5   {
6       int a[3][4] = { {3,5,7,6}, {1,8,2,5},{7,6,2,9}};
7       int *p=&a[0][0];
8       for ( int i = 0; i < 3; i++ ){
9           for ( int j = 0; j < 4; j++ ){
10              cout <<'\t'<< p[i*4+j];
11          }
12          cout<<endl;
13      }
14      return 0;
15  }
```

样例输出

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

第7行的一维指针p指向了二维数组首元素的地址，第10行显示可以用一维的形式遍历二维数组的数据，二维数组的所有元素在一维上是连续的。

知识点

索引	要点	正链	反链
T624	所有数组的物理存储都是一维的，掌握高维数组和一维数组的逻辑对应关系	T623	

2.4 数组作为函数参数

第20行执行会报错，提示错误incompatible types in assignment of : int* to int [5]'，表示类型不匹配；如果改写为a++，会提示错误. lvalue required as increment operand，意思是自增操作需要左值(lvalue)，左值就是可以放在赋值号=左边的标识符，也就是必须要求是变量。引起这两个错误本质上的原因是数组名是常变量，不允许被修改。

```
1   #include<iostream>
2   #include<typeinfo>
3   using namespace std;
4   #define type(obj) typeid(obj).name()
5
6   void dummy(int b0[],int b[],int bb[][4])
7   {
8       cout<<type(b0)<<'\t'<<type(b)<<'\t'<<type(bb)<<endl;
9       cout<<b0[0]<<'\t'<<b<<'\t'<<bb<<endl;
10      cout<<++b0<<'\t'<<++b<<'\t'<<++bb<<endl;
11  }
12
13  int main (  )
14  {
15      int a0=3;
16      int a[5];
17      int aa[5][4];
18      cout<<type(a0)<<'\t'<<type(a)<<'\t'<<type(aa)<<endl;
19      cout<<&a0<<'\t'<<a<<'\t'<<aa<<endl;
20      //a=a+1;//a++;
21      dummy(&a0,a,aa);
22      return 0;
23  }
```

样例输出

```
i       A5_i    A5_A4_i
0x62fe0c        0x62fdf0        0x62fda0
Pi      Pi      PA4_i
3       0x62fdf0        0x62fda0
0x62fe10        0x62fdf4        0x62fdb0
```

2.4 数组作为函数参数

第18行获知a0是一个整数，a是一维数组，aa是二维数组。第8行输出结果说明对应的形参是指针类型。b0是一个整型指针，b也是一个整型指针，bb是一个指向长度为4的数组的指针。b和bb的类型从A变成了P，这也就是意味着数组类型的形参是指针变量，是可修改的。因此第10行可以正常执行。

```
1  #include<iostream>
2  #include<typeinfo>
3  using namespace std;
4  #define type(obj) typeid(obj).name()
5
6  void dummy(int b0[],int b[],int bb[][4])
7  {
8      cout<<type(b0)<<'\t'<<type(b)<<'\t'<<type(bb)<<endl;
9      cout<<b0[0]<<'\t'<<b<<'\t'<<bb<<endl;
10     cout<<++b0<<'\t'<<++b<<'\t'<<++bb<<endl;
11 }
12
13 int main (  )
14 {
15     int a0=3;
16     int a[5];
17     int aa[5][4];
18     cout<<type(a0)<<'\t'<<type(a)<<'\t'<<type(aa)<<endl;
19     cout<<&a0<<'\t'<<a<<'\t'<<aa<<endl;
20     //a=a+1;//a++;
21     dummy(&a0,a,aa);
22     return 0;
23 }
```

样例输出

```
i      A5_i      A5_A4_i
0x62fe0c      0x62fdf0      0x62fda0
Pi      Pi      PA4_i
3      0x62fdf0      0x62fda0
0x62fe10      0x62fdf4      0x62fdb0
```

2.4 数组作为函数参数

- a0和a虽然一个是整数，一个是整型数组，但是都可以**用指针的形式传递给函数**。地址加1后，分别增加了4个字节，但bb是一个二维数组指针，+1后增加了4*4个字节。
- a0虽然是一个整数，也可以被**作为长度为1的数组进行访问**，见第9行。
- 第18行和第9行的输出结果完全相同，因为形参是指针变量，实参将其地址赋值给对应的形参。从实参对形参赋值的角度，就可以理解数组作为形参不能被解析为常变量。

```
1  #include<iostream>
2  #include<typeinfo>
3  using namespace std;
4  #define type(obj) typeid(obj).name()
5
6  void dummy(int b0[],int b[],int bb[][4])
7  {
8      cout<<type(b0)<<'\t'<<type(b)<<'\t'<<type(bb)<<endl;
9      cout<<b0[0]<<'\t'<<b<<'\t'<<bb<<endl;
10     cout<<++b0<<'\t'<<++b<<'\t'<<++bb<<endl;
11 }
12
13 int main (  )
14 {
15     int a0=3;
16     int a[5];
17     int aa[5][4];
18     cout<<type(a0)<<'\t'<<type(a)<<'\t'<<type(aa)<<endl;
19     cout<<&a0<<'\t'<<a<<'\t'<<aa<<endl;
20     //a=a+1;//a++;
21     dummy(&a0,a,aa);
22     return 0;
23 }
```

样例输出

```
i       A5_i    A5_A4_i
0x62fe0c        0x62fdf0        0x62fda0
Pi      Pi      PA4_i
3       0x62fdf0        0x62fda0
0x62fe10        0x62fdf4        0x62fdb0
```

知识点

索引	要点	正链	反链
T625	掌握数组作为函数参数的基本用法，注意数组作为函数参数时实际上就是指针；单个变量也可以被认为是长度为1的数组。	T515	

2.5 C风格字符串与指针

```
1   int ptr_strlen(char* s)
2   {
3       char* p=s;
4       while(*p)    ++p;
5       return p-s;
6   }
```

- 结束字符'\0'的ASCII码为0，代表false，因此第4行不断++向后遍历，直到找到'\0'。
- 第5行中指针p指向'\0'，而s是首字符的地址，二者之间的减法表示了偏移量，即元素的个数。因此p-s代表了字符串的逻辑长度。

```
1   void ptr_strcpy(char* target, char* source)
2   {
3       while(*source)
4           *target++=*source++;
5       *target='\0';              //或*target=*source;
6       // do{
7       //      *target++=*source++;
8       // }while(*source);
9   }
```

- 第4行通过++不断向后偏移，将对应的字符赋值给target
- 第5行时，target指针已经指向尾部，必须给予'\0'，表示字符串的结束。也可以用source进行赋值，因为此时的source正好指向'\0'。
- 第3-5行也可以替换为第6-8行，因为do-while循环先赋值再比较，因此能确保'\0'被赋值。（知识点：T421）

C风格的字符串：**从第一个字符的地址开始向后遍历，到'\0'结束**

2.5 C风格字符串与指针

```
1   int ptr_strcmp(char* str1, char* str2)
2   {
3       while(*str1&&*str1==*str2)
4       {str1++;str2++;}
5       return *str1-*str2;
6   }
```

- 第4行将对应的两个字符作差返回

*str1-*str2=

- 两个字符串关系　两个字符不同
  - 例如abcd和abdc
- 两个字符串相等　同时为'\0'
  - 例如abc和abc
- 两个字符串相等　一个为'\0'
  - 例如abcd和abc

- 第3行添加判断str1是否结束，是为了防止两个字符串相等，然后越过'\0'继续比较无效字符部分。但是因为后面的等于比较，不需要判断str2是否结束。

2.5 C风格字符串与指针

```
1   char* ptr_strcat(char* target, char* source)
2   {
3       char* p=target;
4       while(*target)  target++;
5       do{
6           *target++=*source++;
7       }while(*source);
8       return p;
9   }
```

第3行保留target的首指针，为了最后返回。

第4行先让target指向尾部。

第5-7行将source的内容复制到target的尾部，包括'\0'。

随堂练习

1. 提取一个字符串中的所有数字字符（'0'...'9'）并输出。

样例输入	样例输出
s0d34f0df21gh	034021

2. 提取一个字符串中的所有数字字符（'0'...'9'）将其转换为一个整数输出。

样例输入	样例输出
s0d034f0df21gh	34021

提示：为了防止数据溢出，要进行提前判断，符合要求的才能组合。

知识点

索引	要点	正链	反链
T626	掌握C风格字符串与字符指针的对应关系	T621	

03 堆内存与动态空间空间分配*

堆内存与动态空间分配

内存分配分为**静态分配**和**动态分配**，二者不同分为四个方面：

1）**分配时间**：程序执行分为编辑、编译、连接、运行四个阶段，静态内存在编译开始阶段分配，动态内存在程序运行时分配，因而静态内存不占用CPU资源，而动态内存占用。

2）**分配位置**：静态内存分配在栈区，动态内存分配在堆区。

3）**分配方式**：静态内存由编译器自动分配，自动回收；动态内存由编程者手动分配，手动回收。

4）**分配大小**：静态内存大小明确，分配时就确定了。动态内存无法确定大小，需要指针来指示位置。

堆内存与动态空间分配

1. 局部变量的内存分配

局部变量的内存分配都属于静态内存分配，分配在栈中。因为栈通常比较小，所以局部变量不能申请太大空间。对于多线程的程序，每个线程都有自己的栈内存，栈中的内容不能跨线程调用。

堆内存与动态空间分配

2. 形参与局部变量的生命周期

函数可以定义一些参数（形参），函数体里也可以定义局部变量。对于每一个函数来说，当前函数一旦返回，这些形参和局部变量就出了作用域，占用的内存也就可以安全地释放。**形参和局部变量的生命周期和函数调用的生命周期一致**。一个函数所需要的内存大小在编译时就能知道，如果不考虑编译时优化，每个栈帧的大小等于当前函数的所有形参和局部变量的大小总和，外加一些元数据的大小。

堆内存与动态空间分配

3. 变量名

每个形参和局部变量在栈上的内存地址偏移量也都是固定的，所以只要有个变量名就可以直接获取到这个变量对应的内存地址。实际上，变量名被直接编译成偏移量，编译后的二进制代码不存在变量名。

堆内存与动态空间分配

4. 堆内存与栈内存

堆内存是一种**按需分配**的动态内存管理机制，申请时才分配，申请多少给多少，而**栈内存**只要调了函数就直接把所有本次调用可能需要的内存都进行分配。动态内存分配机制能让数据一直存在，直到手工销毁，也能让多个线程共享同一个内存里的对象，任何线程只要拿到这片内存的地址和大小就能访问。但是由于它的动态，导致数据访问必须通过指针或引用。此外，堆内存分配耗时较多，对于有时间要求的程序，尽量减少使用动态内存分配。

3. 堆内存的动态分配和释放

C++中采用**new和delete**进行堆内存的动态分配和释放，理论上要求new和delete一定要成对出现。因为堆内存分配必须要手动释放，否则会一直存在。如果只分配不释放，可用内存就会越来越少，造成内存泄露。

堆内存与动态空间分配

样例输出

```
7 5 14942544 3
7 0 5
```

```
1   #include <iostream>
2   using namespace std;
3   int main()
4   {
5       int *a = new int;                    //动态申请一个int类型空间
6       int *a1 = new int(5);                //动态申请一个int类型的空间并且初始化为5
7       int *a2 = new int[5];                //动态申请5个int类型数组
8       int *a3 = new int[5]{1, 2, 3, 4, 5}; // C++11开始支持动态数组的初始化
9       *a = 7;
10      cout << *a<<' ' << *a1<<' ' << a2[2]<<' ' << a3[2]<< endl;
11      cout << a[0] <<' '<< a[1] <<' '<< a1[0] << endl;
12      delete a;
13      delete a1;
14      delete[] a2;
15      delete[] a3;
16      return 0;
17  }
```

- 动态内存分配就是分配了一块内存空间，用指针进行访问，第6行的形式表示对单个int类型空间进行初始化为5，而第7行表示同时申请了5个int类型空间。
- 因为a2未初始化，因此输出的值是随机的。

堆内存与动态空间分配

样例输出

```
7 5 14942544 3
7 0 5
```

```
1   #include <iostream>
2   using namespace std;
3   int main()
4   {
5       int *a = new int;                    //动态申请一个int类型空间
6       int *a1 = new int(5);                //动态申请一个int类型的空间并且初始化为5
7       int *a2 = new int[5];                //动态申请5个int类型数组
8       int *a3 = new int[5]{1, 2, 3, 4, 5}; // C++11开始支持动态数组的初始化
9       *a = 7;
10      cout << *a<<' ' << *a1<<' ' << a2[2]<<' ' << a3[2]<< endl;
11      cout << a[0] <<' '<< a[1] <<' '<< a1[0] << endl;
12      delete a;
13      delete a1;
14      delete[] a2;
15      delete[] a3;
16      return 0;
17  }
```

- 第11行可以看出，指针就是按照地址进行访问，因此*a和a[0]的结果是一样的。也可以进行内存地址偏移，但是a+1地址处的内容是未知的，因此输出结果也是未知的。

知识点

索引	要点	正链	反链
T631	理解静态分配和动态分配的区别，掌握动态内存分配和释放的基本写法		T721,T731

THANKS