

C++函数指针总结

6 回复 184 查看



(<https://www.shiyanlou.com/user/8490>) 实验楼管理员



(<https://www.shiyanlou.com/vip>)

2015-11-10 17:37

技术分享 (<https://www.shiyanlou.com/questions/?tag=技术分享>)

文章从指针的概念、类型、值、与数组的关系、与结构类型的关系、与函数的关系……多个方面对C++指针做了较为详细的总结，希望对你有帮助~

分享到微博

全部回答



实验楼管理员 (<https://www.shiyanlou.com/user/8490>)



(<https://www.shiyanlou.com/vip>)

(<https://www.shiyanlou.com/user/8490>)

指针的概念

指针是一个特殊的变量，它里面存储的数值被解释成为内存里的一个地址。要搞清一个指针需要搞清指针的四方面的内容：指针的类型，指针所指向的类型，指针的值或者叫指针所指向的内存区，还有指针本身所占据的内存区。让我们分别说明。

先声明几个指针放着做例子：

例一：

```
int *ptr;
char *ptr;
int **ptr;
int (*ptr)[3];
int *(*ptr)[4];
```

指针的类型

从语法的角度看，你只要把指针声明语句里的指针名字去掉，剩下的部分就是这个指针的类型。这是指针本身所具有的类型。让我们看看例一中各个指针的类型：

```
int *ptr; //指针的类型是int *
char *ptr; //指针的类型是char *
int **ptr; //指针的类型是 int **
int (*ptr)[3]; //指针的类型是 int(*)[3]
int *(*ptr)[4]; //指针的类型是 int *(*)[4]
```

怎么样？找出指针的类型的方法是不是很简单？

指针所指向的类型

当你通过指针来访问指针所指向的内存区时，指针所指向的类型决定了编译器将把那片内存区里的内容当做什么来看待。

从语法上看，你只须把指针声明语句中的指针名字和名字左边的指针声明符*去掉，剩下的就是指针所指向的类型。例如：

```
int *ptr; //指针所指向的类型是int
char *ptr; //指针所指向的类型是char
int **ptr; //指针所指向的类型是 int *
int (*ptr)[3]; //指针所指向的类型是 int() [3]
int *(*ptr)[4]; //指针所指向的类型是 int *() [4]
```

在指针的算术运算中，指针所指向的类型有很大的作用。

指针的类型(即指针本身的类型)和指针所指向的类型是两个概念。当你对C越来越熟悉时，你会发现，把与指针搅和在一起的“类型”这个概念分成“指针的类型”和“指针所指向的类型”两个概念，是精通指针的关键点之一。我看了不少书，发现有些写得差的书中，就把指针的这两个概念搅在一起了，所以看起来前后矛盾，越看越糊涂。

指针的值

指针的值是指针本身存储的数值，这个值将被编译器当作一个地址，而不是一个一般的数值。在32位程序里，所有类型的指针的值都是一个32位整数，因为32位程序里内存地址全都是32位长。

指针所指向的内存区就是从指针的值所代表的那个内存地址开始，长度为sizeof(指针所指向的类型)的一片内存区。以后，我们说一个指针的值是XX，就相当于说该指针指向了以XX为首地址的一片内存区域；我们说一个指针指向了某块内存区域，就相当于说该指针的值是这块内存区域的首地址。

指针所指向的内存区和指针所指向的类型是两个完全不同的概念。在例一中，指针所指向的类型已经有了，但由于指针还未初始化，所以它所指向的内存区是不存在的，或者说是无意义的。

以后，每遇到一个指针，都应该问问：这个指针的类型是什么？指针指向的类型是什么？该指针指向了哪里？

2015-11-10 17:38



实验楼管理员 (<https://www.shiyanlou.com/user/8490>) [vip](https://www.shiyanlou.com/vip)

(<https://www.shiyanlou.com/user/8490>)

指针本身所占据的内存区

指针本身占了多大的内存？你只要用函数sizeof(指针的类型)测一下就知道了。在32位平台里，指针本身占据了4个字节的长度。

指针本身占据的内存这个概念在判断一个指针表达式是否是左值时很有用。

指针的算术运算

指针可以加上或减去一个整数。指针的这种运算的意义和通常的数值的加减运算的意义是不一样的。例如：

例二：

```
char a[20];
int *ptr=a;
...
...
ptr++;
```

在上例中，指针ptr的类型是int*，它指向的类型是int，它被初始化为指向整形变量a。接下来的第3句中，指针ptr被加了1，编译器是这样处理的：它把指针ptr的值加上了sizeof(int)，在32位程序中，是被加上了4。由于地址是用字节做单位的，故ptr所指向的地址由原来的变量a的地址向高地址方向增加了4个字节。

由于char类型的长度是一个字节，所以，原来ptr是指向数组a的第0号单元开始的四个字节，此时指向了数组a中从第4号单元开始的四个字节。

我们可以用一个指针和一个循环来遍历一个数组，看例子：

例三：

```
int array[20];
int *ptr=array;
...
//此处略去为整型数组赋值的代码。
...
for(i=0;i<20;i++)
{
    (*ptr)++;
    ptr++;
}
```

这个例子将整型数组中各个单元的值加1。由于每次循环都将 指针`ptr` 加1，所以每次循环都能访问数组的下一个单元。再看例子：

例四：

```
char a[20];
int *ptr = a;
...
...
ptr += 5;
```

在这个例子中，`ptr` 被加上了5，编译器是这样处理的：将 指针`ptr` 的值加上5乘 `sizeof(int)`，在32位程序中就是加上了5乘4=20。由于地址的单位是字节，故现在的 `ptr` 所指向的地址比起加5后的 `ptr` 所指向的地址来说，向高地址方向移动了20个字节。在这个例子中，没加5前的 `ptr` 指向数组`a`的第0号单元开始的四个字节，加5后，`ptr`已经指向了数组`a`的合法范围之外了。虽然这种情况在应用上会出问题，但在语法上却是可以的。这也体现出了指针的灵活性。

如果上例中，`ptr` 是被减去5，那么处理过程大同小异，只不过`ptr`的值是被减去5乘 `sizeof(int)`，新的`ptr`指向的地址将比原来的 `ptr` 所指向的地址向低地址方向移动了20个字节。

总结一下，一个 指针`ptrold` 加上一个整数`n`后，结果是一个新的 指针`ptrnew`，`ptrnew` 的类型和 `ptrold` 的类型相同，`ptrnew` 所指向的类型和 `ptrold` 所指向的类型也相同。`ptrnew` 的值将比 `ptrold` 的值增加了`n`乘 `sizeof (ptrold所指向的类型)`个字节。就是说，`ptrnew` 所指向的内存区，将比 `ptrold` 所指向的内存区，向高地址方向移动了`n`乘 `sizeof (ptrold所指向的类型)`个字节。一个 指针`ptrold` 减去一个整数`n`后，结果是一个新的指针 `ptrnew`，`ptrnew` 的类型和 `ptrold` 的类型相同，`ptrnew` 所指向的类型和 `ptrold` 所指向的类型也相同。`ptrnew` 的值将比 `ptrold` 的值，减少了`n`乘 `sizeof (ptrold所指向的类型)`个字节，就是说，`ptrnew` 所指向的内存区，将比 `ptrold` 所指向的内存区，向低地址方向移动了`n`乘 `sizeof (ptrold所指向的类型)`个字节。

2015-11-10 17:38



实验楼管理员 (<https://www.shiyanlou.com/user/8490>) (<https://www.shiyanlou.com/vip>)

(<https://www.shiyanlou.com/user/8490>)

运算符 & 和 *

这里 `&` 是取地址运算符，`*` 是...书上叫做“间接运算符”。`&a` 的运算结果是一个指针，指针的类型是`a`的类型加个 `*`，指针所指向的类型是`a`的类型，指针所指向的地址嘛，那就是`a`的地址。`*p` 的运算结果就五花八门了。总之 `*p` 的结果是`p`所指向的东西，这个东西有这些特点：它的类型是`p`指向的类型，它所占用的地址是`p`所指向的地址。

例五：

```
int a=12;
int b;
int *p;
int **ptr;
p=&a; // &a的结果是一个指针，类型是int*，指向的类型是int，指向的地址是a的地址。
*p=24; // *p的结果，在这里它的类型是int，它所占用的地址是p所指向的地址，显然，*p就是变量a。
ptr=&p; // &p的结果是个指针，该指针的类型是p的类型加个*，在这里是int**。该指针所指向的类型是p的类型，这里是int*。该指针所指向的地址就是指针p自己的地址。
*ptr=&b; // *ptr是个指针，&b的结果也是个指针，且这两个指针的类型和所指向的类型是一样的，所以*ptr来给*ptr赋值就是毫无问题的了。
**ptr=34; // **ptr的结果是ptr所指向的东西，在这里是一个指针，对这个指针再做一次*运算，结果就是一个int类型的变量。
```

指针表达式

一个表达式的最后结果如果是一个指针，那么这个表达式就叫指针表达式。下面是一些指针表达式的例子：

例六：

```
int a,b;
int array[10];
int *pa;
pa=&a;//&a是一个指针表达式。
int **ptr=&pa;//&pa也是一个指针表达式。
*ptr=&b;//*ptr和&b都是指针表达式。
pa=array;
pa++;//这也是指针表达式。
```

例七：

```
char *arr[20];
char **parr=arr;//如果把arr看作指针的话，arr也是指针表达式
char *str;
str=*parr;//*parr是指针表达式
str=*(parr+1);//*(parr+1)是指针表达式
str=*(parr+2);//*(parr+2)是指针表达式
```

由于指针表达式的结果是一个指针，所以指针表达式也具有指针所具有的四个要素：指针的类型，指针所指向的类型，指针指向的内存区，指针自身占据的内存。

好了，当一个指针表达式的结果指针已经明确地具有了指针自身占据的内存的话，这个指针表达式就是一个左值，否则就不是一个左值。在例七中，`&a` 不是一个左值，因为它还没有占据明确的内存。`*ptr` 是一个左值，因为 `*ptr` 这个指针已经占据了内存，其实 `*ptr` 就是指针 `pa`，既然 `pa` 已经在内存中有了自己的位置，那么 `*ptr` 当然也有了自己的位置。

2015-11-10 17:39



实验楼管理员 (<https://www.shiyanlou.com/user/8490>) <https://www.shiyanlou.com/vip>

(<https://www.shiyanlou.com/user/8490>)

数组和指针的关系

如果对声明数组的语句不太明白的话，请参阅我前段时间贴出的文章<<如何理解c和c++的复杂类型声明>>。数组的数组名其实可以看作一个指针。看下列：

例八：

```
int array[10]={0,1,2,3,4,5,6,7,8,9},value;
...
...
value=array[0];//也可写成：value=*array;
value=array[3];//也可写成：value=*(array+3);
value=array[4];//也可写成：value=*(array+4);
```

上例中，一般而言数组名 `array` 代表数组本身，类型是 `int[10]`，但如果把 `array` 看做指针的话，它指向数组的第0个单元，类型是 `int *`，所指向的类型是数组单元的类型即 `int`。因此 `*array` 等于0就一点也不奇怪了。同理，`array+3` 是一个指向数组第3个单元的指针，所以 `*(array+3)` 等于3。其它依此类推。

例九：

```
char *str[3]={
    "Hello,this is a sample!",
    "Hi,good morning.",
    "Hello world"
};
char s[80];
strcpy(s,str[0]);//也可写成strcpy(s,*str);
strcpy(s,str[1]);//也可写成strcpy(s,*(str+1));
strcpy(s,str[2]);//也可写成strcpy(s,*(str+2));
```


上例中，`str` 是一个三单元的数组，该数组的每个单元都是一个指针，这些指针各指向一个字符串。把指针数组名 `str` 当作一个指针的话，它指向数组的第0号单元，它的类型是 `char*`，它指向的类型是 `char`。

`*str` 也是一个指针，它的类型是 `char*`，它所指向的类型是 `char`，它指向的地址是字符串 "Hello,this is a sample!" 的第一个字符的地址，即 'H' 的地址。`str+1` 也是一个指针，它指向数组的第1号单元，它的类型是 `char*`，它指向的类型是 `char`。

* (str+1) 也是一个指针，它的类型是 `char*`，它所指向的类型是 `char`，它指向 "Hi,good morning." 的第一个字符 'H'，等等。

2015-11-10 17:40



实验楼管理员 (<https://www.shiyanlou.com/user/8490>)  (<https://www.shiyanlou.com/vip>)

(<https://www.shiyanlou.com/user/8490>)

下面总结一下数组的数组名的问题。声明了一个数组 `TYPE array[n]`，则数组名称 `array` 就有了两重含义：第一，它代表整个数组，它的类型是 `TYPE [n]`；第二，它是一个指针，该指针的类型是 `TYPE*`，该指针指向的类型是 `TYPE`，也就是数组单元的类型，该指针指向的内存区就是数组第0号单元，该指针自己占有单独的内存区，注意它和数组第0号单元占据的内存区是不同的。该指针的值是不能修改的，即类似 `array++` 的表达式是错误的。

在不同的表达式中数组名 `array` 可以扮演不同的角色。

在表达式 `sizeof(array)` 中，数组名 `array` 代表数组本身，故这时 `sizeof` 函数测出的是整个数组的大小。

在表达式 `*array` 中，`array` 扮演的是指针，因此这个表达式的结果就是数组第0号单元的值。`sizeof(*array)` 测出的是数组单元的大小。

表达式 `array+n`（其中 `n=0, 1, 2,`）中，`array` 扮演的是指针，故 `array+n` 的结果是一个指针，它的类型是 `TYPE*`，它指向的类型是 `TYPE`，它指向数组第 `n` 号单元。故 `sizeof(array+n)` 测出的是指针类型的大小。

例十：

```
int array[10];
int (*ptr)[10];
ptr=&array;
```

上例中 `ptr` 是一个指针，它的类型是 `int (*)[10]`，他指向的类型是 `int[10]`，我们用整个数组的首地址来初始化它。在语句 `ptr=&array` 中，`array` 代表数组本身。

本节中提到了函数 `sizeof()`，那么我来问一问，`sizeof(指针名称)` 测出的究竟是指针自身类型的大小呢，还是指针所指向的类型的大小？答案是前者。例如：

```
int (*ptr)[10];
```


则在32位程序中，有：

```
sizeof(int(*)[10])==4
sizeof(int [10])==40
sizeof(ptr)==4
```

实际上，`sizeof(对象)` 测出的都是对象自身的类型的大小，而不是别的什么类型的大小。

2015-11-10 17:40



实验楼管理员 (<https://www.shiyanlou.com/user/8490>)  (<https://www.shiyanlou.com/vip>)

(<https://www.shiyanlou.com/user/8490>)

指针和结构类型的关系

可以声明一个指向结构类型对象的指针。

例十一：

```
struct MyStruct
{
    int a;
    int b;
    int c;
}
MyStruct ss={20,30,40}; //声明了结构对象ss，并把ss的三个成员初始化为20，30和40。
MyStruct *ptr=&ss; //声明了一个指向结构对象ss的指针。它的类型是MyStruct*，它指向的类型是MyStruct。
int *pstr=(int*)&ss; //声明了一个指向结构对象ss的指针。但是它的类型和它指向的类型和ptr是不同的。
```

请问怎样通过指针ptr来访问ss的三个成员变量？

答案：

```
ptr->a;
ptr->b;
ptr->c;
```

又请问怎样通过指针pstr来访问ss的三个成员变量？

答案：

```
*pstr; //访问了ss的成员a。
*(pstr+1); //访问了ss的成员b。
*(pstr+2) //访问了ss的成员c。
```

呵呵，虽然我在我的MSVC++6.0上调式过上述代码，但是要知道，这样使用pstr来访问结构成员是不正规的，为了说明为什么不正规，让我们看看怎样通过指针来访问数组的各个单元：

例十二：

```
int array[3]={35,56,37};
int *pa=array;

通过指针pa访问数组array的三个单元的方法是：
*pa; //访问了第0号单元
*(pa+1); //访问了第1号单元
*(pa+2); //访问了第2号单元
```

从格式上看倒是与通过指针访问结构成员的不正规方法的格式一样。

所有的C/C++编译器在排列数组的单元时，总是把各个数组单元存放在连续的存储区里，单元和单元之间没有空隙。但在存放结构对象的各个成员时，在某种编译环境下，可能会需要字对齐或双字对齐或者是别的什么对齐，需要在相邻两个成员之间加若干个“填充字节”，这就导致各个成员之间可能会有若干个字节的空隙。

所以，在例十二中，即使pstr访问到了结构对象ss的第一个成员变量a，也不能保证 (pstr+1) 就一定能访问到结构成员b。因为成员a和成员b之间可能会有若干填充字节，说不定 *(pstr+1) 就正好访问到了这些填充字节呢。这也证明了指针的灵活性。要是你的目的就是想看看各个结构成员之间到底有没有填充字节，嘿，这倒是个不错的方法。

通过指针访问结构成员的正确方法应该是象例十二中使用指针ptr的方法。

指针和函数的关系

可以把一个指针声明成为一个指向函数的指针。

```
int fun1(char*,int);
int (*pfun1)(char*,int);
pfun1=fun1;
....
....
int a=(*pfun1)("abcdefg",7); //通过函数指针调用函数。
```

可以把指针作为函数的形参。在函数调用语句中，可以用指针表达式来作为实参。

作者：hellodev

文章来源：<http://www.cnblogs.com/ggjucheng/archive/2011/12/13/2286391.html>

2015-11-10 17:41

我要提问

标签

Linux (<https://www.shiyanlou.com/questions/?tag=Linux>) Python (<https://www.shiyanlou.com/questions/?tag=Python>)

C/C++ (<https://www.shiyanlou.com/questions/?tag=C/C++>) 实验环境 (<https://www.shiyanlou.com/questions/?tag=实验环境>)

技术分享 (<https://www.shiyanlou.com/questions/?tag=技术分享>) 功能建议 (<https://www.shiyanlou.com/questions/?tag=功能建议>)

课程需求 (<https://www.shiyanlou.com/questions/?tag=课程需求>) Java (<https://www.shiyanlou.com/questions/?tag=Java>)

其他 (<https://www.shiyanlou.com/questions/?tag=其他>) SQL (<https://www.shiyanlou.com/questions/?tag=SQL>)

NodeJS (<https://www.shiyanlou.com/questions/?tag=NodeJS>) Hadoop (<https://www.shiyanlou.com/questions/?tag=Hadoop>)

常见问题 (<https://www.shiyanlou.com/questions/?tag=常见问题>) Web (<https://www.shiyanlou.com/questions/?tag=Web>)

Shell (<https://www.shiyanlou.com/questions/?tag=Shell>) PHP (<https://www.shiyanlou.com/questions/?tag=PHP>)

Git (<https://www.shiyanlou.com/questions/?tag=Git>) HTML (<https://www.shiyanlou.com/questions/?tag=HTML>)

HTML5 (<https://www.shiyanlou.com/questions/?tag=HTML5>) 信息安全 (<https://www.shiyanlou.com/questions/?tag=信息安全>)

网络 (<https://www.shiyanlou.com/questions/?tag=网络>) GO (<https://www.shiyanlou.com/questions/?tag=GO>)

NoSQL (<https://www.shiyanlou.com/questions/?tag=NoSQL>) 训练营 (<https://www.shiyanlou.com/questions/?tag=训练营>)

Android (<https://www.shiyanlou.com/questions/?tag=Android>) Ruby (<https://www.shiyanlou.com/questions/?tag=Ruby>)

Perl (<https://www.shiyanlou.com/questions/?tag=Perl>)

相关问题

谈Runtime机制和使用的整体化梳理 (<https://www.shiyanlou.com/questions/3010>)

JavaScript：彻底理解同步、异步和事件循环(Event Loop) (<https://www.shiyanlou.com/questions/3009>)

Github上的十大深度学习项目 (<https://www.shiyanlou.com/questions/3000>)

git基础知识整理 (<https://www.shiyanlou.com/questions/2999>)

Linux编程之内存映射 (<https://www.shiyanlou.com/questions/2992>)

动手做实验，轻松学IT。

实验楼-通过动手实践的方式学会IT技术。

公司简介 (<https://www.shiyanlou.com/aboutus>) 联系我们 (<https://www.shiyanlou.com/contact>) 常见问题 (<https://www.shiyanlou.com/faq#howtostart>)
我要开课 (<https://www.shiyanlou.com/labs>) 隐私协议 (<https://www.shiyanlou.com/privacy>) 会员条款 (<https://www.shiyanlou.com/terms>)
友情链接 (<https://www.shiyanlou.com/friends>)
站长统计 (http://www.cnzz.com/stat/website.php?web_id=5902315) 蜀ICP备13019762号 (<http://www.miibeian.gov.cn/>)



微信

