4.++p是左值，p++是右值

3、为什么i++不能作左值?

我们来看++i和i++的实现就明白了。

```
// 前缀形式:
int& int::operator++() //这里返回的是一个引用形式，就是说函数返回值也可以作为一个左值使用
{//函数本身无参，意味着是在自身空间内增加1的
  *this += 1;  // 增加
  return *this;  // 取回值
}

//后缀形式:
const int int::operator++(int) //函数返回值是一个非左值型的，与前缀形式的差别所在。
{//函数带参，说明有另外的空间开辟
  int oldValue = *this;  // 取回值
  ++(*this);  // 增加
  return oldValue;  // 返回被取回的值
}
```

简单得到理解，就是i++返回的是一个临时变量，函数返回后不能被寻址得到，它只是一个数据值，而非地址，因此不能作为左值。

```
// i++:
{
int tmp;
tmp=i;
i=i+1;
return tmp;
}

// ++i:
{
i=i+1;
return i;
}
```

5.cin不是function

7.s2 cppreference上没有这个初始化方式-> warning

s4 48个1

8.'+' 跟随前面的类型

10.B:引用的目的: 更改，减少不必要的拷贝构造

E. 函数返回值类型少了个const

F.第一题: const char [5]

11.B new 会调用构造函数，malloc不会

12.print_array_cpp 传入的是一个长度为10 的数组，而print_array_c的参数会退化成为int*类型的指针

14.signed int=int,

int [10] 退化成int*

15.b是a的reference

b,d都是a的引用=>int* c指向的是a

16.有同学反应说>>会报错，较早的编译器确实会这样，但新的编译器已经优化了

17.顶层const和底层const的问题

19.C里面NULL就是0

c++ standard规定只能是最普通的0，计算出来的0都是不可以的

std::move() 函数将一个左值强制转化为右值引用，以用于移动语义。

移动语义，允许直接转移对象的资产和属性的所有权，而在参数为右值时无需复制它们。

换一种说法就是，std::move() 将对象的状态或者所有权从一个对象转移到另一个对象，只是转移，没有内存的搬迁或者内存拷贝。

因此，通过std::move()，可以避免不必要的拷贝操作。

比如下面这个例子，原str（lvalue值）被moved from之后值被转移，所以为空字符串。

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

int main()
{
   string str = "hello";
   cout << "before str: " << str << endl;

   vector<string> vstr;
   vstr.emplace_back(std::move(str));
   cout << "after  str: " << str << endl;

   return 0;
}
```