CS100 Recitation 7

GKxx

Contents

- 引用
- 左值和右值
- `std::vector`
- `new` 和 `delete`

引用 (Reference)

动机

练习：使用基于范围的 `for` 语句遍历一个字符串，将大写改为小写

动机

练习：使用基于范围的 `for` 语句遍历一个字符串，将大写改为小写

```
for (char c : str)
  c = std::tolower(c);
```

这样写是**不行的**：`for (char c : str)` 相当于让 `char c` 依次成为 `str` 中的每个字符的拷贝，在 `c` 上修改不会影响 `str` 的内容。

如同

```
for (std::size_t i = 0; i != str.size(); ++i) {
  char c = str[i];
  c = std::tolower(c);
}
```

动机

练习：使用基于范围的 `for` 语句遍历一个字符串，将大写改为小写

```
for (char &c : str)
  c = std::tolower(c);
```

`char &c` 定义了一个**绑定到 `char` 的引用**，我们让 `c` 依次绑定到 `str` 中的每个字符。

如同

```
for (std::size_t i = 0; i != str.size(); ++i) {
  char &c = str[i];
  c = std::tolower(c); // effectively str[i] = std::tolower(str[i]);
}
```

引用即别名

```
int ival = 42;
int &iref = ival;
++iref;  // ival becomes 43
iref = 50; // ival becomes 50
std::cout << ival << std::endl; // 50
std::cout << iref << std::endl; // 50
int &iref2 = iref; // Equivalent to int &iref2 = ival;
                   // iref2 also bounds to ival.
```

可以写 `Type &r`，也可以 `Type& r`

- 但 `Type& r1, r2, r3;` 只有 `r1` 是引用。
- 如果想定义多个引用，需要 `Type &r1, &r2, &r3;`

引用必须初始化（即在定义时就指明它绑定到谁），并且这个绑定关系不可修改。

引用只能绑定到实际的对象

变量、数组、指针等都是实际存在的对象，但“引用”本身只是一个傀儡，是一个实际存在的对象的“别名”。

- 可以定义绑定到数组的引用：

```
int a[10];
int (&ar)[10] = a;
ar[0] = 42; // a[0] = 42
```

- 可以定义绑定到指针的引用：

```
int *p;
int *&pr = p;
pr = &ival; // p = &ival;
```

引用只能绑定到实际的对象

变量、数组、指针等都是实际存在的对象，但“引用”本身只是一个傀儡，是一个实际存在的对象的“别名”。

- 不能定义绑定到引用的引用。
- 也不能定义指向引用的指针，因为指针也必须指向实际的对象。

传引用参数 (pass-by-reference)

可以定义绑定到数组的引用：

```
void print_array(int (&array)[10]) {
  for (int x : array)           // 基于范围的 for 语句可以用来遍历数组！
    std::cout << x << ' ';
  std::cout << std::endl;
}
int a[10], ival = 42, b[20];
print_array(a);     // Correct
print_array(&ival); // Error!
print_array(b);     // Error!
```

这个 `array` 真的是（绑定到了）一个数组，而不是指向数组首元素的指针。

- `array` 只能绑定到 `int[10]`，其它牛鬼蛇神都会 Error（C 做不到这一点）。

传引用参数 (pass-by-reference)

定义一个函数，接受一个字符串，输出其中的大写字母。

```
void print_upper(std::string str) {
  for (char c : str)
    if (std::isupper(c))
      std::cout << c;
  std::cout << std::endl;
}
```

传引用参数 (pass-by-reference)

```
void print_upper(std::string str) {
  for (char c : str)
    if (std::isupper(c))
      std::cout << c;
  std::cout << std::endl;
}
std::string s = some_very_long_string();
print_upper(s);
```

传参的过程中发生了形如 `std::string str = s;` 的**拷贝初始化**。如果 `s` 很长，这将非常耗时。

传引用参数 (pass-by-reference)

以**引用**的方式传参，避免拷贝：

```
void print_upper(std::string &str) {
  for (char c : str)
    if (std::isupper(c))
      std::cout << c;
  std::cout << std::endl;
}
std::string s = some_very_long_string();
print_upper(s); // std::string &str = s;
```

传引用参数 (pass-by-reference)

以**引用**的方式传参，避免拷贝：

```
void print_upper(std::string &str) {
  for (char c : str)
    if (std::isupper(c))
      std::cout << c;
  std::cout << std::endl;
}
std::string s = some_very_long_string();
print_upper(s); // std::string &str = s;
```

但这样有问题：

```
print_upper("Hello"); // Error
```

左值 (lvalue) 和右值 (rvalue)

左值和右值

一个表达式在被使用时，有时我们使用的是它代表的**对象**，有时我们仅仅是使用了那个对象的**值**。

- `str[i] = ch` 中，我们使用的是表达式 `str[i]` 所代表的**对象**
- `ch = str[i]` 中，我们使用的是表达式 `str[i]` 所代表的对象的**值**。

左值和右值

一个表达式在被使用时，有时我们使用的是它代表的**对象**，有时我们仅仅是使用了那个对象的**值**。

- `str[i] = ch` 中，我们使用的是表达式 `str[i]` 所代表的**对象**
- `ch = str[i]` 中，我们使用的是表达式 `str[i]` 所代表的对象的**值**。

一个表达式本身带有**值类别** (value category) 的属性：它要么是左值，要么是右值

- 左值：它代表了一个实际的对象
- 右值：它仅仅代表一个值

左值和右值

一个表达式本身带有**值类别** (value category) 的属性：它要么是左值，要么是右值

- 左值：它代表了一个实际的对象
- 右值：它仅仅代表一个值

哪些表达式能代表一个实际的对象？

哪些表达式产生的仅仅是一个值？

左值和右值

一个表达式本身带有**值类别** (value category) 的属性：它要么是左值，要么是右值

- 左值：它代表了一个实际的对象
- 右值：它仅仅代表一个值

哪些表达式能代表一个实际的对象？

- `*p`, `a[i]`

哪些表达式产生的仅仅是一个值？

- `a + b`, `&val`, `cond1 && cond2` 等等。

* **“左值”和“右值”这两个名字是怎么来的？**

左值和右值

在 C 中，左值可以放在赋值语句的左侧，右值不能。

但在 C++ 中，二者的区别远没有这么简单。

目前已经见过的返回左值的表达式：`*p`，`a[i]`

特别地：在 C++ 中，前置递增/递减运算符返回**左值**，`++i = 42` 是合法的。

赋值表达式返回左值：`a = b` 的返回值是 `a` 这个对象。

- 试着解释表达式 `a = b = c` ？

左值和右值

在 C 中，左值可以放在赋值语句的左侧，右值不能。

但在 C++ 中，二者的区别远没有这么简单。

目前已经见过的返回左值的表达式：`*p`，`a[i]`

特别地：在 C++ 中，前置递增/递减运算符返回**左值**，`++i = 42` 是合法的。

赋值表达式返回左值：`a = b` 的返回值是 `a` 这个对象。

- 赋值运算符**右结合**，表达式 `a = b = c` 等价于 `a = (b = c)`
- 先执行 `b = c`，然后相当于 `a = b`。

左值和右值

右值仅仅代表一个值，不代表一个实际的对象。常见的右值有**表达式执行产生的临时对象**和**字面值**。

```
std::string fun(); // a function that returns a std::string object
std::string a = fun();
```

- 函数调用 `fun()` 生成的临时对象是**右值**。
- 特别的例外：**字符串字面值** `"hello"` **是左值**，它其实是真实存在于内存中的对象。
    - 相比之下，整数字面值 `42` 仅仅产生一个临时对象，是右值。

引用只能绑定到左值

```
int ival = 42;
int &iref = 42;          // Error
int &iref2 = ival;       // Correct
int &iref3 = ival + 42; // Error
int fun();
int &iref4 = fun();      // Error
```

C++11 引入了所谓的“右值引用”，我们在介绍**移动**的时候再讲。一般来说，“引用”指的是“左值引用”。

引用是实际对象的别名，所以引用也是左值

```
int arr[10];
int &subscript(int i) { // function returning int&
  return arr[i];
}
subscript(3) = 42; // Correct.
int &ref = subscript(7); // Correct. ref bounds to arr[7]
```

Reference-to-`const`

类似于“指向常量的指针”（即带有“底层 `const`”的指针），我们也有“绑定到常量的引用”

```
int ival = 42;
const int cival = 42;
const int &cref = cival; // Correct
const int &cref2 = ival; // Also correct.
int &ref = cival;  // Error: Casting-away low-level const is not allowed.
int &ref2 = cref;  // Error: Casting-away low-level const is not allowed.
int &ref3 = cref2; // Error: Even though cref2 bounds to non-const ('ival'),
                   // this is still casting-away low-level const.
```

一个 reference-to-`const` **自认为自己绑定到 `const` 对象**，所以不允许通过它修改它所绑定的对象的值，也不能让一个不带 `const` 的引用绑定到它。（不允许“去除底层 `const`”）

Reference-to-`const`

指针既可以带顶层 `const`（本身是常量），也可以带底层 `const`（指向的东西是常量），但引用**不谈**“顶层 `const`”。

- 即，只有“绑定到常量的引用”。引用本身不是对象，不谈是否带 `const`。
- 从另一个角度讲，引用本身一定带有“顶层 `const`”，因为绑定关系不能修改。
- 在不引起歧义的情况下，通常用**常量引用**这个词来代表“绑定到常量的引用”。

Reference-to-`const`

特殊规则：常量引用可以绑定到右值：

```
const int &cref = 42; // Correct
int fun();
const int &cref2 = fun(); // Correct
int &ref = fun(); // Error
```

当一个常量引用被绑定到右值时，实际上就是让它绑定到了一个临时对象。

- 这是合理的，反正你也不能通过常量引用修改那个对象的值

Pass-by-reference-to-`const`

```
void print_upper(std::string &str) {
  for (char c : str)
    if (std::isupper(c))
      std::cout << c;
  std::cout << std::endl;
}
print_upper("Hello"); // Error
```

当我们传递 `"Hello"` 给 `std::string` 参数时，实际上发生了一个由 `const char [6]` 到 `std::string` 的**隐式转换**，这个隐式转换产生**右值**，无法被 `std::string&` 绑定。

```
const std::string s = "hello";
print_upper(s); // Error: Casting-away low-level const
```

不带底层 `const` 的引用无法绑定到 `const` 对象。

Pass-by-reference-to-`const`

将参数声明为**常量引用**，既可以避免拷贝，又可以允许传递右值

```
void print_upper(const std::string &str) {
  for (char c : str)
    if (std::isupper(c))
      std::cout << c;
  std::cout << std::endl;
}
std::string s = some_very_long_string();
print_upper(s); // const std::string &str = s;
print_upper("Hello"); // const std::string &str = "Hello";, Correct
```

也可以传递常量对象：

```
const std::string s = "hello";
print_upper(s); // OK
```

Pass-by-reference-to-`const`

将参数声明为**常量引用**，既可以避免拷贝，又可以允许传递右值，也可以传递常量对象，也可以**防止你不小心修改了它**。

在 C++ 中声明函数的参数时，**尽可能使用常量引用**（如果你不需要修改它）。

（如果仅仅是 `int` 或者指针这样的内置类型，可以不需要常量引用）

Pass-by-reference-to-`const`

练习：编写一个函数，接受一个字符串，倒序输出其中所有的小写字母。

Pass-by-reference-to-`const`

练习：编写一个函数，接受一个字符串，倒序输出其中所有的小写字母。

```
void print_lower_reversed(const std::string &str) {
  for (int i = str.size() - 1; i >= 0; --i)
    if (std::islower(str[i]))
      std::cout << str[i];
  std::cout << std::endl;
}
```

这里的 `i` 必须带符号！

总结

引用：

- 引用即别名
- 引用必须绑定到实际的对象

左值和右值：

- 左值是实际的对象，右值通常是一些临时对象或者字面值
- 使用左值是使用实际的对象，使用右值仅仅是使用那个值
    - 例如：你可以修改一个对象，但无法修改一个值
- 常见的返回左值的表达式：`*p`，`a[i]`，`++i`，`a = b`
- 引用只能绑定到左值

总结

常量引用：

- 常量引用可以绑定到左值，也可以绑定到右值
- 常量引用“自以为”自己绑定到了常量，所以自发地保护它所绑定到的对象
- `const` 是一把锁，**只允许加锁，不允许解锁**
- pass-by-reference-to-`const` 的好处：
  - 避免拷贝
  - 允许传递右值和常量
  - 避免一不小心修改这个对象

真正的“值类别”

（语言律师需要掌握）

C++ 中的表达式依值类别被划分为如下三种：

英文	中文	has identity?	can be moved from?
lvalue	左值	yes	no
xvalue (expired value)	亡值	yes	yes
prvalue (pure rvalue)	纯右值	no	yes

lvalue + xvalue = glvalue（广义左值），xvalue + prvalue = rvalue（右值）

- 所以实际上“左值是实际的对象”是不严谨的，右值也可能是实际的对象（xvalue）

`std::vector`

定义在标准库文件 `<vector>` 中

真正好用的“动态数组”

创建一个 `std::vector` 对象

`std::vector` 是一个**类模板**，只有给出了模板参数之后才成为一个真正的类型。

```
std::vector<int> vi;          // An empty vector of ints
std::vector<std::string> vs; // An empty vector of strings
std::vector<double> vd;       // An empty vector of doubles
```

不同模板参数的 `vector` 是**不同的类型**。

创建一个 `std::vector` 对象

```
std::vector<int> v{2, 3, 5, 7}; // A vector of ints,
                                // whose elements are {2, 3, 5, 7}.
std::vector<int> v2 = {2, 3, 5, 7}; // Equivalent to ↑

std::vector<std::string> vs{"hello", "world"}; // A vector of strings,
                           // whose elements are {"hello", "world"}.
std::vector<std::string> vs2 = {"hello", "world"}; // Equivalent to ↑

std::vector<int> v3(10); // A vector of ten ints, all initialized to 0.
std::vector<int> v4(10, 42); // A vector of ten ints, all initialized to 42.
```

`vector<T> v(n)` 这种构造方式会将 `n` 个元素都**值初始化**（类似于 C 中的"空初始化"），而不是得到一串未定义的值！

创建一个 `std::vector` 对象

```
std::vector<int> v{2, 3, 5, 7};
std::vector<int> v2 = v; // v2 is a copy of v
std::vector<int> v3(v);  // Equivalent
std::vector<int> v4{v};  // Equivalent
```

去年 CS100 一直到期末居然还有人用循环一个元素一个元素拷贝 `vector`，**太愚蠢了！**

```
std::vector<std::vector<int>> v;
```

“二维 `vector`”，也就是“`vector` of `vector`”，当然也是可以的。

C++17 CTAD

Class Template Argument Deduction：只要你给出了足够的信息，编译器可以自动推导元素的类型！

```
std::vector v{2, 3, 5, 7}; // vector<int>
std::vector v2{3.14, 6.28}; // vector<double>
std::vector v3(10, 42); // vector<int>
std::vector v4(10); // Error: cannot deduce template argument type
```

`std::vector` 的大小

`v.size()` 和 `v.empty()`

```
std::vector v{2, 3, 5, 7};
std::cout << v.size() << std::endl;
if (v.empty()) {
  // ...
}
```

清空 `std::vector`

`v.clear()`。不要写愚蠢的 `while (!v.empty()) v.pop_back();`

向 `std::vector` 添加元素

`v.push_back(x)` 将元素 `x` 添加到 `v` 的末尾

```
int n;
std::cin >> n;
std::vector<int> v;
for (int i = 0; i != n; ++i) {
  int x;
  std::cin >> x;
  v.push_back(x);
}
```

删除 `std::vector` 最后一个元素

`v.pop_back()`

练习：将末尾的偶数删掉，直到末尾是奇数为止

```
while (!v.empty() && v.back() % 2 == 0)
  v.pop_back();
```

`v.back()`：获得末尾元素**的引用**（这意味着什么？）

`v.back()` 和 `v.front()`

分别获得最后一个元素、第一个元素的**引用**。

“引用”意味着你可以通过这两个成员函数修改它们：

```
v.front() = 42;
++v.back();
```

`v.back()`, `v.front()`, `v.pop_back()` 在 `v` 为空的情况下是 undefined behavior，而且实际上是**严重的运行时错误**。

基于范围的 `for` 语句

遍历一个 `std::vector`，同样可以使用基于范围的 `for` 语句：

```
std::vector<int> vi = some_values();
for (int x : vi)
  std::cout << x << std::endl;
std::vector<std::string> vs = some_strings();
for (const std::string &s : vs) // use reference-to-const to avoid copying
  std::cout << s << std::endl;
```

练习：使用基于范围的 `for` 语句，将一个 `vector<string>` 中的每个字符串的大写字母打印出来。

基于范围的 `for` 语句

练习：使用基于范围的 `for` 语句，将一个 `vector<string>` 中的每个字符串的大写字母打印出来。

```
for (const std::string &s : vs) {
  for (char c : s)
    if (std::isupper(c))
      std::cout << c;
  std::cout << std::endl;
}
```

使用下标访问

可以使用 `v[i]` 来获得第 `i` 个元素

- `i` 的有效范围是 $[0, N)$，其中 `N = v.size()`
- 越界访问是**未定义行为**，并且通常是**严重的运行时错误**。
- `std::vector` 的下标运算符 `v[i]` **并不检查越界**，目的是为了保证效率。
  - 事实上标准库容器的大多数操作都没有对合法性进行检查，为了效率。
- 一种检查越界的下标是 `v.at(i)`，它会在越界时抛出 `std::out_of_range` 异常。
  - 不妨自己试一试。
  - C++ 中的异常处理？看看我的视频

接口的统一性

事实上 `std::string` 也有 `.at()`, `.front()`, `.back()`, `.push_back(x)`, `.pop_back()`, `.clear()` 等函数。C++ 标准库的各种设施是讲究统一性的。

完整列表

练习：实现 Python 的 `rstrip` 函数，接受一个 `std::string`，返回它删去末尾的连续空白后的结果。

接口的统一性

练习：实现 Python 的 `rstrip` 函数，接受一个 `std::string`，返回它删去末尾的连续空白后的结果。

```
std::string rstrip(std::string str) {
  while (!str.empty() && std::isspace(str.back()))
    str.pop_back();
  return str;
}
```

在 C++17 下，这个 `return str;` 是不会产生拷贝的，不必担心。（看看我的视频）

`std::vector` 的增长策略

考虑像这样连续 `push_back` `n` 次得到一个 `vector` 的代码：

```
std::vector<int> v;
for (int i = 0; i != n; ++i)
  v.push_back(i);
```

`vector` 是如何做到快速增长的？

`std::vector` 的增长策略

假设现在有一片动态分配的内存，长度为 `i`。

当第 `i+1` 个元素到来时，朴素做法：

1. 分配一片长度为 `i+1` 的内存
2. 将原有的 `i` 个元素拷贝过来
3. 将新的元素放在后面
4. 释放原来的那片内存

但这需要拷贝 `i` 个元素。`n` 次 `push_back` 总共就需要 $\sum_{i=0}^{n-1} i = O\left(n^2\right)$ 次拷贝！

`std::vector` 的增长策略

假设现在有一片动态分配的内存，长度为 `i`。

当第 `i+1` 个元素到来时，

1. 分配一片长度为 `2*i` 的内存
2. 将原有的 `i` 个元素拷贝过来
3. 将新的元素放在后面
4. 释放原来的那片内存

而当第 `i+2`, `i+3`, ..., `2*i` 个元素到来时，我们不需要分配新的内存，也不需要拷贝任何对象！

`std::vector` 的增长策略

$0 \to 1 \to 2 \to 4 \to 8 \to 16 \to \cdots$

假设 $n = 2^m$，那么总的拷贝次数就是 $\sum_{i=0}^{m-1} 2^i = O(n)$，**平均**（“均摊”）一次 `push_back` 的耗时是 $O(1)$（常数），可以接受。

使用 `v.capacity()` 来获得它目前所分配的内存的实际容量，看看是不是真的这样。

- 注意：这仅仅是一种可能的策略，标准并未对此有所规定。

* **看看 hw4/prob2** `attachments/testcases/vector.c`

`std::vector` 动态增长带来的影响

我们已经看到，改变 `vector` 的大小可能会导致它所保存的元素“搬家”，这会使得所有指针、引用、迭代器失效。

- 最直接的影响：下面的代码是 undefined behavior，因为基于范围的 `for` 语句本质上依赖于迭代器。

```
for (int i : vec)
  if (i % 2 == 0)
    vec.push_back(i + 1);
```

不要在用基于范围的 `for` 语句遍历容器的同时改变容器的大小！

`new` 和 `delete`（初步）

`new` 表达式

动态分配内存，**并构造对象**

```
int *pi1 = new int;      // 动态创建一个默认初始化的 int
int *pi2 = new int();    // 动态创建一个值初始化的 int
int *pi3 = new int{};    // 同上，但是更 modern
int *pi4 = new int(42); // 动态创建一个 int，并初始化为 42
int *pi5 = new int{42}; // 同上，但是更 modern
```

对于内置类型：

- **默认初始化** (default-initialization)：就是未初始化，具有未定义的值
- **值初始化** (value-initialization)：类似于 C 中的“空初始化”，是各种零。

`new[]` 表达式

动态分配“数组”，**并构造对象**

```
int *pai1 = new int[n];            // 动态创建了 n 个 int，默认初始化
int *pai2 = new int[n]();          // 动态创建了 n 个 int，值初始化
int *pai3 = new int[n]{};          // 动态创建了 n 个 int，值初始化
int *pai4 = new int[n]{2, 3, 5}; // 动态创建了 n 个 int，前三个元素初始化为 2,3,5
                                   // 其余元素都被值初始化（为零）
                                   // 如果 n<3，抛出 std::bad_array_new_length 异常
```

对于内置类型：

- **默认初始化** (default-initialization)：就是未初始化，具有未定义的值
- **值初始化** (value-initialization)：类似于 C 中的“空初始化”，是各种零。

`delete` 和 `delete[]` 表达式

销毁动态创建的对象，并释放其内存

```
int *p = new int{42};
delete p;
int *a = new int[n];
delete[] a;
```

- `new` 必须对应 `delete`，`new[]` 必须对应 `delete[]`，否则是 **undefined behavior**
- 忘记 `delete`：内存泄漏

一一对应，不得混用

违反下列规则的一律是 undefined behavior:

- `delete ptr` 中的 `ptr` 必须等于某个先前由 `new` 返回的地址
- `delete[] ptr` 中的 `ptr` 必须等于某个先前由 `new[]` 返回的地址
- `free(ptr)` 中的 `ptr` 必须等于某个先前由 `malloc`, `calloc`, `realloc` 或 `aligned_alloc` 返回的地址。

`new`/`delete` vs `malloc`/`free`

C++ 的对象模型比 C 复杂得多，而 `new`/`delete` 也比 `malloc`/`free` 做了更多的事：

- `new`/`new[]` 表达式会**先分配内存，然后构造对象**。对于类类型的对象，它可能会调用一个合适的**构造函数**。
- `delete`/`delete[]` 表达式会**先销毁对象，然后释放内存**。对于类类型的对象，它会调用**析构函数**。

在 C++ 中，非必要不手动管理内存

- 当你需要创建“一列数”、“一列对象”，或者“一张表”、“一个集合”时，**优先考虑标准库容器等设施**，例如 `std::string`，`std::vector`，`std::deque` (双端队列)，`std::list` / `std::forward_list` (链表)，`std::map` / `std::set` (红黑树)，`std::unordered_map` / `std::unordered_set` (哈希表)
- 当你需要创建单个对象时，应该优先考虑**智能指针** (`std::shared_ptr`，`std::unique_ptr`，`std::weak_ptr`)
- 只有在特殊情况下（例如手搓一个标准库没有的数据结构，并且对效率有极高的要求），使用 `new` / `delete` 来管理动态内存
- 当你对于内存分配本身也有特殊的要求时，才需要使用 C 的内存分配/释放函数，但通常也是用它们来定制 `new` 和 `delete`

特别提一下：`NULL` vs `nullptr`

- `NULL` 是一个用 `#define` 定义的宏，可能的定义有 `0`，`(long)0`，`(void *)0` 等等
- C++ 不会将 `NULL` 定义为 `(void *)0`，因为 C++ 不允许 `void *` 向其它指针类型的隐式转换。
- C++ 中的 `NULL` 大概率是 `0`，`(long)0` 或者 `(long long)0`，**但这是一个整数而非指针**，会使得一些类型推导和重载决议发生错误。
- 从语言设计上来说，为了支持 `NULL`，C++ 也不得不引入一些丑陋的特殊规则。

更好的空指针：`nullptr`

`nullptr` 是真正的“空指针”，于 C++11 引入，并即将加入 C23。

`nullptr` 具有独一无二的类型 `std::nullptr_t`，无需破坏原有的类型规则，在重载决议时也会优先匹配指针而非整数。

* **在 C++ 中，请使用 `nullptr` 表示空指针，而不是 `NULL` 或者数 `0`。**